ISSN 1213-6972 *Volume 21, Number 3, 2013*

Journal of WSCG

An international journal of algorithms, data structures and techniques for computer graphics and visualization, surface meshing and modeling, global illumination, computer vision, image processing and pattern recognition, computational geometry, visual human interaction and virtual reality, animation, multimedia systems and applications in parallel, distributed and mobile environment.

Vaclav Skala – Union Agency

Journal of WSCG

Editor-in-Chief: Vaclav Skala
c/o University of West Bohemia
Faculty of Applied Sciences
Univerzitni 8
CZ 306 14 Plzen
Czech Republic
http://www.VaclavSkala.eu http://www.wscg.eu

Managing Editor: Vaclav Skala

Printed and Published by:
Vaclav Skala - Union Agency
Na Mazinach 9
CZ 322 00 Plzen
Czech Republic

Hardcopy: ***ISSN 1213 – 6972***
CD ROM: ***ISSN 1213 – 6980***
On-line: ***ISSN 1213 – 6964***

Journal of WSCG

Editor-in-Chief

Vaclav Skala

c/o University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering
Univerzitni 8
CZ 306 14 Plzen
Czech Republic

http://www.VaclavSkala.eu
Journal of WSCG URLs: http://www.wscg.eu or http://wscg.zcu.cz/jwscg

Editorial Advisory Board
MEMBERS

WSCG 2013

Board of Reviewers

Agathos, Alexander
Assarsson, Ulf
Ayala, Dolors
Backfrieder, Werner
Barbosa, Joao
Barthe, Loic
Battiato, Sebastiano
Benes, Bedrich
Benger, Werner
Bilbao, Javier,J.
Biri, Venceslas
Birra, Fernando
Bittner, Jiri
Bosch, Carles
Bourdin, Jean-Jacques
Brun, Anders
Bruni, Vittoria
Buehler, Katja
Bulo, Samuel Rota
Cakmak, Hueseyin
Camahort, Emilio
Casciola, Giulio
Cline, David
Coquillart, Sabine
Cosker, Darren
Daniel, Marc
Daniels, Karen
de Geus, Klaus
de Oliveira Neto, Manuel Menezes
Debelov, Victor
Drechsler, Klaus
Durikovic, Roman
Eisemann, Martin
Erbacher, Robert
Feito, Francisco
Ferguson, Stuart
Fernandes, Antonio
Fuenfzig, Christoph
Gain, James
Galo, Mauricio
Gobron, Stephane
Grau, Sergi
Gudukbay, Ugur
Guthe, Michael
Hansford, Dianne
Haro, Antonio
Hasler, Nils
Hast, Anders
Hernandez, Benjamin
Hernandez, Ruben Jesus Garcia
Herout, Adam
Herrera, Tomas Lay
Hicks, Yulia
Hildenbrand, Dietmar
Hinkenjann, Andre
Chaine, Raphaelle
Choi, Sunghee
Chover, Miguel
Chrysanthou, Yiorgos
Chuang, Yung-Yu
Iglesias, Jose,A.
Ihrke, Ivo
Iwasaki, Kei
Jato, Oliver
Jeschke, Stefan
Jones, Mark
Juan, M.-Carmen
Kämpe, Viktor
Kanai, Takashi
Kellomaki, Timo
Kim, H.
Klosowski, James
Kolcun, Alexej
Krivanek, Jaroslav
Kurillo, Gregorij
Kurt, Murat
Kyratzi, Sofia
Larboulette, Caroline
Lee, Jong Kwan Jake
Liu, Damon Shing-Min
Lopes, Adriano
Loscos, Celine
Lutteroth, Christof
Maciel, Anderson
Mandl, Thomas
Manzke, Michael
Marras, Stefano
Masia, Belen
Masood, Syed Zain
Max, Nelson
Melendez, Francho
Meng, Weiliang
Mestre, Daniel,R.
Metodiev, Nikolay Metodiev
Meyer, Alexandre
Molina Masso, Jose Pascual
Molla, Ramon
Montrucchio, Bartolomeo
Morigi, Serena
Muller, Heinrich
Munoz, Adolfo
Murtagh, Fionn
Okabe, Makoto
Oyarzun, Cristina Laura
Pan, Rongjiang
Papaioannou, Georgios
Paquette, Eric
Pasko, Galina
Patane, Giuseppe
Patow, Gustavo
Pedrini, Helio
Pereira, Joao Madeiras
Peters, Jorg

Pina, Jose Luis
Platis, Nikos
Post, Frits,H.
Puig, Anna
Rafferty, Karen
Renaud, Christophe
Reshetouski, Ilya
Reshetov, Alexander
Ribardiere, Mickael
Ribeiro, Roberto
Richardson, John
Rojas-Sola, Jose Ignacio
Rokita, Przemyslaw
Rudomin, Isaac
Sacco, Marco
Salvetti, Ovidio
Sanna, Andrea
Santos, Luis Paulo
Sapidis, Nickolas,S.
Savchenko, Vladimir
Seipel, Stefan
Sellent, Anita
Shesh, Amit
Sik-Lanyi, Cecilia
Sintorn, Erik
Skala, Vaclav
Slavik, Pavel
Sochor, Jiri
Sourin, Alexei
Sousa, A.Augusto
Sramek, Milos
Stroud, Ian
Subsol, Gerard
Sundstedt, Veronica
Szecsi, Laszlo
Teschner, Matthias
Theussl, Thomas
Tian, Feng
Tokuta, Alade
Torrens, Francisco
Trapp, Matthias
Tytkowski, Krzysztof
Umlauf, Georg
Vasa, Libor
Vergeest, Joris
Vitulano, Domenico
Vosinakis, Spyros
Walczak, Krzysztof
WAN, Liang
Wu, Shin-Ting
Wuensche, Burkhard,C.
Wuethrich, Charles
Xin, Shi-Qing
Xu, Dongrong
Yoshizawa, Shin
Yue, Yonghao
Zalik, Borut
Zara, Jiri
Zemcik, Pavel
Zhang, Xinyu
Zhao, Qiang
Zheng, Youyi
Zitova, Barbara
Zwettler, Gerald

Journal of WSCG

Vol.21, No.3

Contents

Distributed VFX Architecture for SPH Simulation

Dieter Morgenroth
HDM Stuttgart
Nobelstraße 10
D 70569 Stuttgart

Daniel Weiskopf
University of Stuttgart
Visualisierungsinstitut
Allmandring 19
D 70569 Stuttgart

Bernhard Eberhardt
HDM Stuttgart
Nobelstraße 10
D 70569 Stuttgart

ABSTRACT

We propose an approach to simulating and rendering physically based fluid effects in a VFX production environment using a client/server architecture that is practical for distributed simulation resources and that can be seamlessly integrated into commercial 3D animation packages. The fluid simulation implements smoothed particle hydrodynamics (SPH). We extend the concept of surface particles by introducing blind particles that facilitate efficient direct raytracing of isosurfaces. We evaluate the performance of our approach with local simulation on CPUs and GPUs and distributed GPU simulation. We demonstrate the integration into the animation package 3ds Max and the VRay raytracer. The usability of the VFX production pipeline is assessed by a user study with VFX professionals and animation experts.

Keywords

Smoothed particle hydrodynamics, VFX pipeline, direct raytracing of implicit surfaces

1 INTRODUCTION

An important, however expensive and time-consuming part in CGI/VFX production is physically based simulation. There are many different and competing approaches, with respective advantages and disadvantages. In addition, to experiment with new VFX ideas, the integration of research results into VFX pipelines of production houses is a continuing challenge because the production pipeline adds requirements to the system that may turn good theory into complicated implementation. Many leading VFX production houses like ILM or CA Scanline use in-house solutions for physical simulation. A few specialized software vendors such as Next Limit Technologies offer solutions like Realflow [1] that couple to established animation packages. Recently, software packages that started as in-house solutions have been emerging on the market, like the fluid simulation software Naiad from Autodesk.

All current solutions, including the above, are either directly integrated into the animation package as extensions like Glu3D [2] from 3DAliens for Autodesk's 3ds Max, where the simulation can be run directly in the base package, or the simulation is run in an external application like Realflow and the results are then imported into the animation package as baked simulation data. A good example of a typical pipeline integration in a VFX environment is described by Lagergren [3], whose fluid simulation with a GPU implementation targets the production renderer in the SideFx Houdini system.

Here, we propose a client/server architecture for distributed VFX simulation that can be seamlessly used within an existing VFX pipeline. Similar to other system papers like the one by Parker et al. [4], we describe the design choices that have to be made for the integration of recent research advances into a production pipeline.

The proposed design pattern of this paper is applicable to many areas in the field of simulation. We have chosen fluid simulation as a concrete example because of its high relevance for special effects and since it also affects the rendering aspect of the production pipeline, thus demonstrating how we can address the integration of software components in all relevant stages of the pipeline.

We have designed a client/server fluid simulation system that allows artists to interactively setup simulations and change parameters inside their familiar animation package while the system executes the numerical calculation for the simulation on a remote system with specialized hardware.

We have chosen a particle-based method for fluid simulation because all major 3D packages have built-in particle systems that can handle simulated point data. The coupling to existing particle systems has the advantage that a variety of tools are already available to further

use of the simulation data, for example, for triggering the creation of splash particles.

Our second contribution is an enhanced direct raytracing technique for isosurfaces that extends the concept of surface particles by introducing blind particles. This reduces the amount of particles to be evaluated for rendering and the network bandwidth required for data exchange, which is the main bottleneck in client/server systems. As a side effect, this also speeds up direct raytracing methods since the rays can skip empty space inside the fluid.

As a case study, we have implemented our approach into the 3D package 3ds Max and used external computing capacity based on NVidia GPUs for the server side. With a user study with VFX professionals, we have evaluated the workflow in a production environment.

2 PHYSICAL BACKGROUND

For this paper, we use smoothed particle hydrodynamics (SPH) [5] for fluid simulation. A good overview of particle-based simulation methods can be found elsewhere [6].

Our implementation uses the kernel functions for density, viscosity, and pressure from Mueller et al. [7]. It employs constant particle size, but adaptive methods could be included as well [8, 9].

For tension forces, we apply the approach by Becker and Teschner [10] using cohesion forces. Integration of the dynamics of the particle system employs explicit first-order Euler integration because it delivers sufficient numerical quality at high computational speed. However, higher-order schemes could be easily used instead, if necessary. To achieve incompressibility, we use predictive-corrective incompressible SPH (PCISPH) [11].

3 SYSTEM ARCHITECTURE

Our focus is good usability of efficient fluid simulation in a VFX environment. To overcome the computational bottleneck of simulating a huge amount of particles on the workstation itself, we outsource the computing power to an external GPU server. Still, the results of the simulation steps should be visible as soon as possible and steering the simulation parameters should reside in the 3D scene in the main package.

To achieve this level of interactivity, a client/server architecture is used; see Figure 1. The client resides in the commercial 3D package and sends simulation parameters to the server while requesting simulation results for specific frames. Decoupling of simulation computation and scene management has the benefit that several users can share specialized hardware. Recently,

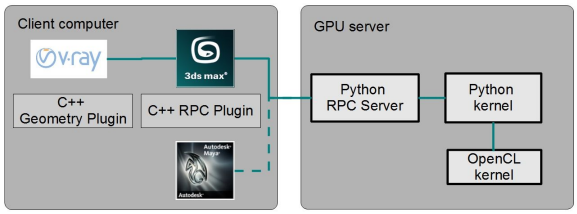


Figure 1: Architecture overview: outsourcing simulation tasks to remote hosts.

GPU blades like the Tesla workstations became affordable for smaller studios. Outsourcing simulation work to these specialized servers is a logical consequence.

The server runs on an external hardware and identifies the user and scene. The user can change parameters for the scene in the 3D package and then start a simulation for a specific frame range. The server will run the simulation in a background thread and store the results in a local disk cache. In the host application, the simulation is an operator in the particle simulation framework. The operator holds the simulation parameters and the current state of the particles as input parameters for the simulation and receives the particles over the network. We compute the SPH-related forces in the simulation server and allow the host application to add forces from the integrated particle system and then do one integration step for all forces. This opens the way for a variety of effects. With this seamless integration, for example, the interaction of the SPH simulation with forces coming from an inverse kinematics (IK) skeleton that is driven with motion capture data is possible.

In our current implementation, we integrated the client into 3ds Max. However, other host systems such as Autodesk Maya or SideFx Houdini would work equally well. For maximum flexibility, we integrated the client into the Particle Flow particle system as an operator using the Particle Flow SDK.

The communication between the client and the server is implemented via remote procedure calls (RPCs). The main problem in implementing a remote SPH fluid simulation is the huge amount of data to be transferred. A typical SPH-VFX shot uses several million SPH particles. The simulation data has to be stored at least for every frame. If the host system needs subframe precision to calculate secondary effects like e.g. additional spray particles, the data has to be saved even more frequently. This computed data has to be stored and streamed over the network. Hence, hard disk space and bandwidth rapidly become the critical factors and need careful decision making regarding file format and message protocol.

As network message protocol, we chose the msgpack [12] protocol, which is a bandwidth-efficient protocol for binary data for which implementations in C++ and Python exist. In addition, it allows

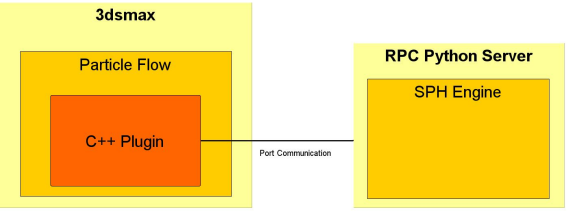


Figure 2: Remote procedure call architecture.

compressed communication with little overhead. The integration with 3ds Max was done using its event-driven particle system Particle Flow. Particle Flow can display all particle emitters and operators in a node-based editor visualizing a graph of nodes. Several data channels can flow through this graph. With the Particle Flow SDK, new channels and operators can be added to the system using C++. Figure 2 illustrates the remote procedure call architecture.

For our implementation, we added a new operator that holds the msgpack-rpc C++ client. This operator then sends remote procedure calls to the server. The client can request the particle count for a specific time. Our system exchanges position, velocity, and color for each particle. The color value will be written to the color channel in 3ds Max. We use it to send the particle type and particle density. We use float32 values for each array element. This leads to memory requirements of $(9 \times 4) = 36$ bytes per particle, which comes to about 3.6 MB per 100 k particles. This can be compressed to about 60 percent using zip compression. The operator unpacks the data and then sets both the particle count of the current Particle Flow graph and the positions and information about the particle type.

All performance-critical modules of the simulation were written as parallel code in OpenCL to run on both multi-CPU and GPU setups.

4 SIMULATION

The Particle Flow framework employs an event-driven model, based on the concept of events and operators. Operators describe and modify particle properties such as speed and direction over a period of time. Individual operators can be combined into groups called events. Each operator provides a set of parameters that define particle behaviors during the event. Particle Flow continuously evaluates each operator and updates the particle system accordingly. Particles can be sent from event to event using tests. Tests let you connect events together in series. A test can check for certain properties of the particles. Particles that pass the test move on to the next event, while those that do not meet the test criteria remain in the current event [13]. We implemented the simulation code as an operator of Particle Flow. This decouples the SPH simulation from the computation of other forces that may act upon the particles.

Particle Flow has its own integration routine. To avoid 'double' integration a Particle Flow node can overwrite the default integrator with a custom implementation. By this, we can integrate the particles in our operator and make sure that boundary conditions and collisions are handled properly. By reading out the acceleration channel we can use the Particle Flow forces to affect the simulation. The acceleration due to the Particle Flow forces changes the velocity on the client side while the SPH acceleration is added on the server. The mass is defined as constant for all particles; therefore, we can simply add the accelerations:

$$a_{total} = a_{SPH} + a_{ParticleFlow} \tag{1}$$

We adopted the PCI SPH technique [11] with strategies for GPU implementation according to Goswami et al. [14]. PCI SPH offers good time/visual quality ratio, high robustness, and easy implementation. However, the choice of simulation algorithm should not affect the system architecture. One of the benefits of our approach is that maintaining and changing the simulation implementation is decoupled and transparent: it can be done without changes to the local software setup of the client workstations as long as the parameter set stays the same.

4.1 Neighbor Search

SPH simulation is well suited for a parallel implementation since the particles can easily be distributed to the computing units. Efficient neighbor search for each particle is the challenging part for parallel implementations. Our decision for an acceleration structure was based on the requirement that we need a memory-efficient structure that can run on the GPU. We also wanted a concept that can be extended to a multi-GPU scenario similar to the multi-GPU work by Valdez-Balderas et al. [15].

Ihmsen et al. [16] compared different neighbor search data structures for multi-core architectures with shared memory and found that using a regular grid for neighborhood search with zCurve sorting to be the fastest solution for parallel implementations. For sorting the particles into the cells, we use the algorithm described by Goswami et al. [14]. Each particle is assigned a single integer hash value based on its cell coordinates. For better memory caching, a space-filling curve numbering is employed instead of a simple hash value. After a key/value sort of this array and the particle indexes, all particles that belong to the same cell can be found in a continuous order. By executing binary search in the hash array, the particles for a particular cell can be found.

4.2 Collision Detection

Collision geometry is sent to the server with a RPC call. We send the polygon positions and a global transform matrix of the object for each frame. We generate a hash value for the polygons of each object based on the vertex positions. The polygons are cached on the simulation server and are only updated if the hash value changes. To handle collisions we intersect the particle trajectories with the triangles in both the PCI prediction step and the final integration step. If a collision is found, we correct the position of the particle to the intersection point and set its velocity to zero. The additional pressure correction iterations of the PCI method propagate the pressure back into the fluid and lead to visually satisfying results, according to our experiences. Therefore, more sophisticated boundary handling methods are not needed for our use cases.

4.3 Blind Particles

To reduce the amount of data required for rendering, we introduce the use of blind particles. Only particles near the surface contribute to the surface generation and the rendering process. The particles inside the volume are not of any interest for visualization. This observation was used before to accelerate raytracing. We also exploit this fact for optimized disk caching and reduction of bandwidth requirements. Recent works [17, 14, 18] identify the surface particles from the simulation and only use those for surface generation. [18] use surface particles for surface generation in a tesselation process based on Marching Cubes, but not for directly raytracing the surface. [18] store the surface information on the grid that they use for their Marching Cubes algorithm while we are tagging the underlying particles with the particle type property directly.

However, identifying only surface particles is not sufficient when used with a production renderer. Figure 3 (a) illustrates this problem. In this example, the inner red surface only emerges because the inner particles were deleted. This problem arises for commercial raytracers, because the core components of the raytracing algorithm cannot be changed by plugins. Therefore, we cannot decide if the surface is valid or not by the algorithm, especially if camera rays that start inside the fluid should be possible. A ray that enters a fluid and passes

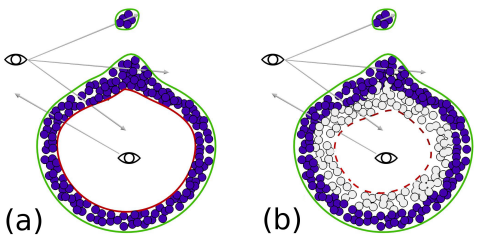


Figure 3: Surface generation without and with blind particles.

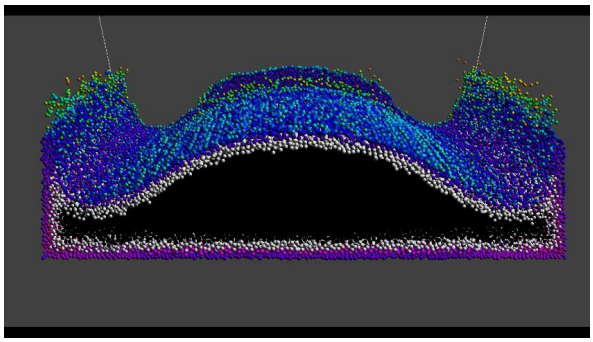


Figure 4: SPH particles color-coded by particle type. Black: particles to be omitted, white: blind particles, colored: surface particles color-coded by pressure.

the surface particles will generate a surface on the inside of the boundary particle hull. For the ray, it could also be a small splash particle or a thin fluid stream.

To solve this problem, we introduce blind particles. We not only identify the surface particles but also a thin layer of particles tagged as 'blind' around them. See Figure 3 (b). Blind particles prevent the generation of the inner surface. The rest of the particles is only relevant during simulation. Omitting particles would normally create new surfaces inside the fluid where they are missing. The blind particles will tell the surface generation algorithm to not generate a surface where their field value would normally create an isosurface. For identifying the surface particles, we adopt the method described by Goswami et al. [14]. A blind particle is tagged as blind if at least one neighbor in a threshold distance was tagged as a boundary particle. Figure 4 shows a typical scenario with blind particles, surface particles, and particles that are to be omitted.

4.4 Implementation Details

Similar to Goswami et al. [14], we use the zCurve approach for parallel implementation on the GPU. We temporarily store the neighborhood of each particle in memory for each timestep to reuse the information in the prediction/correction loop.

For the simulation code, we chose OpenCL over CUDA because of the ability to run it efficiently on CPUs as well. For the host code, we chose the Python OpenCL integration [19]. We experienced that kernels run at the same speed independent of whether host code is written in C++ or Python.

The advantage of a Python/OpenCL-based framework is optimal portability. The code ran on all combinations of Windows/Linux and GPU/CPU with only little modifications. However, some optimizations that would be possible for GPU only code were sacrificed for CPU compatibility. For example, the use of shared local memory was avoided in the kernel code.

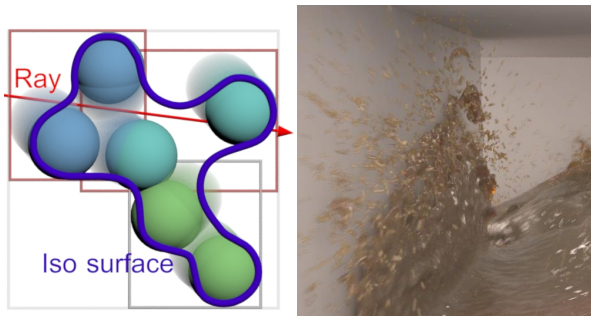


Figure 5: BVH for motion-blurred isosurface.

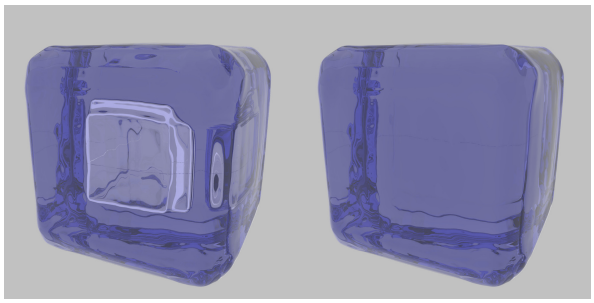


Figure 6: Rendering of a particle cube where inner particles were omitted: (left) without blind particles and (right) with blind particles.

5 RENDERING

Our contribution for the rendering of fluid surfaces is the integration of current fluid rendering algorithms in a VFX pipeline. Often, rendering is a step separate from animation and simulation. To adapt to this separation we designed the rendering routines as a geometry plugin for 3ds Max from Autodesk and the raytracer VRay from Chaosgroup. This renderer is available for all major 3D packages and is widely used in VFX companies in Europe. In general, there are two ways that are used to render fluid surfaces. One approach is to generate a triangle mesh that is then rendered with the standard triangle pipeline. Akinci et al. [18] proposed an efficient parallel implementation for the surface reconstruction. The other approach is direct raytracing of the surface where the intersection with the surface is found for every ray [20]. [21] accelerated direct rendering of fluid surfaces on the GPU by sampling the particle values to a perspective grid. We implemented a surface reconstruction algorithm and a direct raytracing routine and compare both solutions.

In the preparation phase, we fill the particles into a bounding volume hierarchy (BVH), taking the radius of each particle into account. Our implementation differs from other current implementations [22] regarding that if motion blur is required, we use both the position at the beginning of the motion blur interval and at the end to define the leaf nodes. Each particle carries velocity information; the ray request from the raytracing system has time information. With this information, each sample can calculate the exact positions of the particles in the intersected BVH leaves that were generated for the full frame length. Figure 5 shows how the BVH is used for rendering motion-blurred isosurfaces.

As a first step, we find the intersected leaf nodes. We then sort those nodes by their intersection point along the ray. We then march along the ray from the first intersection point into bounding boxes and evaluate a level-set function for the particles in the leaf node. We implemented both the simple Blinn blob and the surface function according to Zhu and Bridson [23]. If the sign of the result changes from one evaluation to the next, then the surface lies between those marching steps. The accurate position is found via binary search.

We omit the intersection if most particles involved in the calculation of the level set are flagged as blind particles. This robustly handles all secondary rays like shadow rays, reflection/refraction rays, and rays for indirect illumination. Also rays that start inside the fluid are handled correctly. Figure 6 compares rendering with and without blind particles.

Direct raytracing is very fast for opaque materials. For ray marching near the boundaries, only outer leaf nodes of the BVH have to be considered and only a few particles add to the level-set function. Once the first intersection is found, the function is finished and may return. The slightly longer raytracing times compared to intersection with polygons are compensated with the much faster preparation times per frame, since no marching cubes [24] or similar meshing phases are needed for explicit isosurface extraction. Especially in cases where huge water masses are simulated, but the camera captures just a fraction of them, the direct rendering method can speed up rendering substantially. An important advantage of the direct raytracing approach is the handling of 3D motion blur. Mesh-based rendering methods have to calculate multiple meshes per frame to obtain clean motion blur in order to compensate for changes in the mesh topology. Modern raytracers can render motion blur based on velocity information per vertex, but this approach can not consider topology changes e.g. merging of fluid drops. For direct raytracing, it does not matter if the topology changes inside the time frame of the motion blur. In Figure 5, the isosurface for a single point in time is drawn as seen by a specific ray sample. For each sample, we calculate the exact position of the surfaces at the requested point in time.

However, the advantage of faster rendering times is lost for transparent materials like water. Here, ray marching has to continue through the material. Also reflections inside the fluid add to the rendering times. Here, the caching strategies VRay provides for polygons outperform direct raytracing. To allow for polygon-based ray-

Scene Stepsize dt	Dambreak 20k 0.01s	Dambreak 125k 0.01s	Dambreak 1.25M 0.01s
Local Intel i7	0h:01m:27s	0h:13m:29s	2h:54m:08s
Local NVidia GTX 570M	0h:00m:48s	0h:05m:06s	1h:09m:28s
Local NVidia GTX 680	0h:00m:19s	0h:02m:16s	0h:29m:34s
Server/client GTX 680	0h:00m:20s	0h:02m:19s	0h:30m:17s

Table 1: Simulation time for 100 frames.

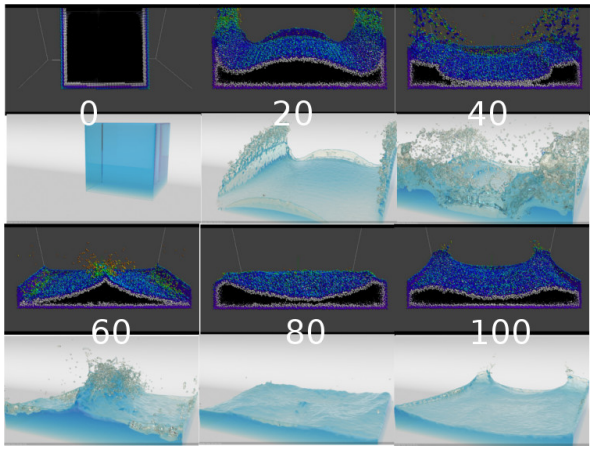


Figure 7: Different time steps of our dambreak simulation.

tracing we also implemented a multi-threaded marching cubes algorithm to create polygons.

6 RESULTS

6.1 Performance Results

We provide performance numbers for the simulation and the rendering components of our system. All tests use a dambreak simulation designed by us. Figure 7 illustrates different time steps of the simulation; also see the accompanying video. We use different volume quantities for the SPH simulation keeping the kernel radius at 4 cm and time step at 0.01 s (with 20 k, 125 k, or 1.25 M particles). The test environment was a mobile workstation equipped with an NVidia GTX 570 M GPU and Intel I7 CPU (2.0 GHz) and a desktop workstation with an NVidia GTX 680. In the client/server configuration, the mobile workstation was the client with 3ds Max and the desktop workstation was the simulation server. The network was a 1 GBit Ethernet connection.

Table 1 documents the simulation times for the different hardware platforms and SPH quantities. The simulation times include transfer of the particle data between client and server for each frame and, therefore, may be slower than times presented in other SPH real-time papers. However, the difference between running the server on the same machine (“local" in Table 1) and running over the network was only about 2 percent. The advantage of having access from the mobile workstation to the computing power of the GTX 680 card in the desktop workstation, which resulted in about twice the speed, by far compensated the communication overhead. Especially, time-critical productions may benefit from the flexible use of external compute resources.

Scene		Dambreak 20 k	Dambreak 200 k
Marching Cubes	opaque	0:36	2:46
	transp.	0:39	2:27
Direct raytracing	opaque	0:18	2:01
	transp.	0:55	6:18
Direct raytracing blind	opaque	0:18	1:57
	transp.	0:55	4:43

Table 2: Rendering times (min:sec).

Table 2 shows the rendering times on the mobile workstation for the small and medium-sized dambreak simulations. The rendering times were recorded for frame 40 of the simulation, where a large number of active particles are present. As shown in Figure 8, there are other time steps of the simulation that have much fewer active particles. In those cases, savings from direct raytracing are even more pronounced than for frame 40 of the animation.

Finally, Figure 9 compares rendering times between the blind particle method and the conventional method. The direct raytracing implementation is especially helpful in the setup phase for low-resolution test rendering. Especially for small cropped render tests without antialiasing, direct raytracing outperforms methods that need preparation steps with meshing. This allows fast iterations in the lighting phase.

Longer preparation times for polygon generation pay off in more complex scenes with multiple ray bounces. For each ray sample, the scalar field of the isosurface has to be evaluated multiple times for the binary search in order to find the intersection point. This is a costly operation. For small resolutions without antialiasing and only single ray bounces, this is still faster than surface reconstruction. Antialiasing schemes dramatically increase the rendering time for direct raytracing while only moderately increase the rendering time for polygons. This can be seen in Table 2 for frame 40 of our dambreak simulation. All tests were performed with an image resolution of 1280×720 pixel and fixed-rate antialiasing. Our conclusion is that direct raytracing should be used in the setup phase, when small resolution test renders are made or in situations where only

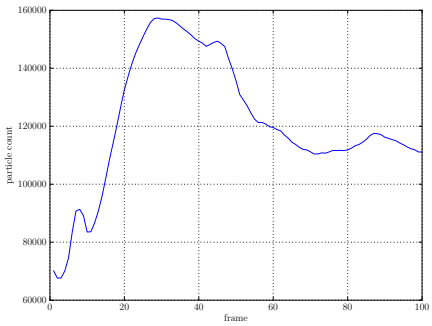


Figure 8: Particle count of active particles (boundary and blind particles) of the 200 k dambreak simulation.

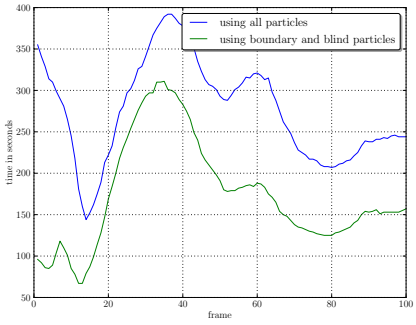


Figure 9: Rendering times for the blind particle and conventional methods for the 200 k dambreak simulation.

portions of the simulation volume are seen by the camera.

Our geometry plugin supports all VRay render elements. Being able to deliver all requested render element passes is an important factor for successful integration in today's production pipelines.

6.2 User Study

We assessed the usability and effectiveness of our system conducting a user study with VFX professionals. The study is mainly of qualitative nature, accompanied by quantitative and qualitative questionnaires. In addition to gathering information to improve the usability of our system (as part of a formative process), we wanted to test two hypotheses: "SPH solver integration into Particle Flow improves the workflow in 3ds Max" (Hypothesis 1) and "Direct raytracing is suitable for VFX production" (Hypothesis 2).

Our study was designed to test realistic work environments and tasks. Therefore, we recruited specialists for fluid-related VFX. Due to the highly specialized user group, the number of participants was small (three). Therefore, we have chosen a qualitative user study design: we used the think-aloud method [25] in addition to the questionnaires to obtain maximum feedback from each individual. According to Nielsen [26], three expert users can be sufficient for a think-aloud study. A screen and audio recording was later transcribed into text and selected comments were then categorized into the phases 'initial setup', 'tweaking simulation', 'tweaking surfacing', 'tweaking render settings', and placed in a review document.

All participants had more than 3 years of professional experience with 3D software and spent more than 6 hours per day working with 3ds Max. All participants were working on fluid-related VFX projects in their jobs at the time of the user study. Since VFX professionals with a fluid background and 3ds Max experience are scattered over the world, the user study was performed remotely with screen sharing sessions. The participants temporarily installed our plugin on their workstations. After an introductory session of about 30 minutes, we asked them to design a dambreak-like animation. The task was easily explained and can be solved in the short amount of time available. As an additional requirement, they were asked to add additional forces from 3ds Max to the simulation like e.g. a vortex to explore the coupling with 3ds Max forces. The participants were instructed to think aloud while they work to protocol their first impressions.

After the test, we asked the participants to fill in an online survey with questions about the usability of the system. Included in these were questions about different areas of the system like *"Integrating fluid solver in Particle Flow allows me to achieve a greater variety of effects than a standalone solver."* and the 10 standard System Usability Scale (SUS) questions [27]. The duration of the test was between 30 and 45 minutes.

All participants agreed that the overall workflow is better if the fluid simulation is integrated in the 3D content creation software in contrast to standalone fluid simulation applications. The usability was also rated very good in the SUS questionnaire. This is attributable to the fact that the solver blended into the interface for which they were experts. In particular, all users appreciated the flexibility that the integration into Particle Flow offered. We also asked about the subjective opinion on the visual quality of direct raytracing of fluid blobs in contrast to meshing approaches. All agreed that direct raytracing offers superior image quality; no participant considered it slow. All would consider it for their next project. The detailed questions and results of the questionnaires can be found in the supplementary material.

The participants were also asked for unstructured feedback. Some of the representative feedback includes: "The integration of an SPH solver into a particle system is basically interesting. It often happens in daily work that you have to add small fluid simulations on top of existing particle simulations, where you don't want to setup a big system. For example liquid spurts in battle scenes, where a complete fluid simulation would

be too much, but still small effects are needed." Or: "This should actually in theory kill the performance because we are raytracing into an isosurface and that is something you shouldn't do. Sure enough it is going slightly slower but testament to the quality of the mesh it's intersecting – even at the low settings – it is really not." Or: "It is really a dream to be able actually stop a render this quickly and go back to your settings, and change them, tweak them, press f9 to re-render and it is there.... Seriously, on a daily basis I have to wait 15 minutes between stopping a render and releasing all memory possible from the computer it takes minutes and then I make the one small change like 'I need two more of this' and press f9 and wait 10 minutes before the fist bucket is on screen."

In summary, the user study provides a preliminary indication that our approach provides a useful integration of SPH simulation in the VFX workflow (Hypothesis 1) and that direct raytracing can be suitable for certain aspects of VFX production (Hypothesis 2).

7 CONCLUSION AND FUTURE WORK

Our approach shows that external and distributed computing resources can be integrated in the established 3D workflow of VFX production companies seamlessly using commercial software packages allowing interactive sessions. In our user study, we confirmed the good utility of our approach for domain experts. We have presented an approach to reducing the memory footprint of particle caches without visual difference. This is especially important if the data has to be transferred over network. The idea of blind particles is independent from the simulation concept used and can be easily integrated in existing pipelines with savings in both rendering time and storage requirements. The performance tests confirmed that the distributed simulation leads to negligible communication overhead.

In future work, our implementation could be extended to a multi-GPU system to be employed in scalable hardware environments such as GPU clusters on the server side. Our generic client/server architecture could be extended to other fields of physically based simulation. Similarly, other commercial 3D packages could be integrated with our system.

8 ACKNOWLEDGMENTS

This work was partly supported by "Kooperatives Promotionskolleg Digital Media" at Stuttgart Media University and the University of Stuttgart.

9 REFERENCES

[1] Next Limit. Realflow product website. `http://www.realflow.com`.

[2] 3D Aliens. Glu3d product website. `http://3daliens.com/joomla`.

[3] M. Lagergren. GPU accelerated SPH simulation of fluids for VFX. Report LiU-ITN-TEK-A–10/044–SE, Dept. Sci. Tech., Linköping University, 2010.

[4] E. G. Parker and J. F. O'Brien. Real-time deformation and fracture in a game environment. In *Proc. ACM SIGGRAPH/Eurograph. Symp. Comput. Anim.*, pages 156–166, 2009.

[5] J. J. Monaghan. Smoothed particle hydrodynamics. *Ann. Rev. Astron. Astrophys.*, 30:543–574, 1992.

[6] B. Adams and M. Wicke. Meshless approximation methods and applications in physics based modeling and animation. In *Eurograph. 2009 Tutorials*, pages 213–239, 2009.

[7] M. Mueller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proc. ACM SIGGRAPH/Eurograph. Symp. Comput. Anim.*, pages 154–159, 2003.

[8] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. Adaptively sampled particle fluids. *ACM Trans. Graph.*, 26(3):48, 2007.

[9] H. Yan, Z. Wang, J. He, Xi Chen, C. Wang, and Q. Peng. Real-time fluid simulation with adaptive SPH. *Comput. Anim. Virt. Worlds*, 20:417–426, 2009.

[10] M. Becker and M. Teschner. Weakly compressible SPH for free surface flows. In *Proc. ACM SIGGRAPH/Eurograph. Symp. Comput. Anim.*, pages 209–217, 2007.

[11] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible SPH. *ACM Trans. Graph.*, 28(3):40:1–40:6, 2009.

[12] Furuhashi S. msgpack website. `http://msgpack.org`.

[13] Autodesk. Autodesk 3ds Max userguide. `http://docs.autodesk.com/3DSMAX/15/ENU/3ds-Max-Help/index.html`.

[14] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proc. ACM SIGGRAPH/Eurograph. Symp. Comput. Anim.*, pages 55–64, 2010.

[15] D. Valdez-Balderas, J. M. Domínguez, B. D. Rogers, and A. J. C. Crespo. Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters. *J. Parallel Distrib. Comput*, 2012. doi 10.1016/j.jpdc.2012.07.010.

[16] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner. A parallel SPH implementation

on multi-core CPUs. *Comput. Graph. Forum*, 30(1):99–112, 2011.

[17] Y. Zhang. Adaptive sampling and rendering of fluids on the GPU. In *Proc. Symp. Point-Based Graph.*, pages 137–146, 2008.

[18] G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner. Parallel surface reconstruction for particle-based fluids. *Comput. Graph. Forum*, 31(6):1797–1809, 2012.

[19] A. Kloeckner, N. Pinto, Y. Lee, B.C. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. *CoRR*, abs/0911.3456, 2009.

[20] J. C. Hart. Ray tracing implicit surfaces. *ACM SIGGRAPH 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces*, pages 1–16, 1993.

[21] R. Fraedrich, S. Auer, and R. Westermann. Efficient high-quality volume rendering of sph data. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1533–1540, 2010.

[22] O. Gourmel, A. Pajot, M. Paulin, L. Barthe, and P. Poulin. Fitted BVH for fast raytracing of metaballs. *Comput. Graph. Forum*, 29(2):281–288, 2010.

[23] Y. Zhu and R. Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, 2005.

[24] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Comput. Graph.*, 21(4):163–169, 1987.

[25] C. H. Lewis. Using the “Thinking Aloud" method in cognitive interface design. Tech. report RC-9265, IBM, 1982.

[26] J. Nielsen. Estimating the number of subjects needed for a thinking aloud test. *Intl. J. Human-Comput. Stud.*, 41(3):385–397, 1994.

[27] J. Brooke. SUS: a quick and dirty usability scale. In P. W. Jordan, B. Weerdmeester, A. Thomas, and I. L. Mclelland, editors, *Usability Evaluation in Industry*. Taylor and Francis, 1996.

Screen Space Foam Rendering

Nadir Akinci Alexander Dippel Gizem Akinci Matthias Teschner
nakinci,dippela,gakinci,teschner@informatik.uni-freiburg.de
University of Freiburg
Georges Koehler Allee 052
79110 Freiburg Germany

ABSTRACT

We present a method for the efficient rendering of large scale particle-based foam data in screen space using a GPU based rendering pipeline. Our approach employs a multi-pass rendering technique to imitate some of the effects that are commonly accomplished by using expensive ray-tracing based methods. We demonstrate through different scenarios that our pipeline is able to produce convincing foam renderings for large scale scenarios and it has a significant performance advantage compared to using ray-casting techniques for rendering such particle data.

Keywords

Rendering, Fluids, Foam, Particles

1 INTRODUCTION

Foam is a complex phenomenon whose behavior and appearance is challenging to simulate in computer graphics. When viewed from a close distance, foam is composed of many air bubbles sticking to each other. It can occur inside most fluids as a result of trapped air. One can observe milky white foam caused by dashing waves on seashores. For most semi-transparent materials, it is an interesting observation that, even though the underlying material may have a color, the foam usually looks whitish to the viewer. The reason for this behavior is that the foam is composed of thin films of fluid containing air. As the number of such thin films increase per unit volume, all incoming light is reflected without allowing any light to penetrate beneath it. This optical phenomenon makes the foam look brighter than the material itself, to the point that it looks almost white. This paper focuses on the efficient rendering of such white foam by approximating some important effects in screen space, that are otherwise time consuming to compute in a physically correct way. Our technique is specifically useful for complex large-scale scenarios, where large amount of foam data need to be rendered. In the remainder of this section, we first summarize the existing works about GPU accelerated rendering of fluid data (Sec. 1.1), foam simulation and rendering (Sec.1.2) and then highlight our contribution (Sec. 1.3).

1.1 GPU Rendering of Fluids

For non-interactive applications, fluid surfaces are generally visualized by triangulating the isosurface of the particle data (e.g. [ZB05, YT10, AIAT12]) and then rendering the resultant mesh using ray-tracing based techniques to produce convincing results. For real-time applications, the computational overhead of those approaches remains too high. Therefore, for the efficient GPU accelerated visualization of fluid surfaces, several methods have been proposed in the recent years, e.g., using screen space surface construction [MSD07, FAW10], height field techniques [CM10] and methods that are based on particle splatting [vdLGS09, BSW10]. Even though foam is actually composed of the molecules of the underlying fluid, its characteristic appearance requires it to be handled using different rendering approaches, which will be explained in the next section.

1.2 Foam Simulation and Rendering

In computer graphics, foam generation techniques are used to enhance the realism of existing fluid simulations. High quality foam simulation and rendering techniques are commonly encountered in movies [GLR+06, BSK+07] and in commercial fluid simulation and visualization packages [hyb11]. In those works, however, the underlying foam generation and rendering stages are usually described briefly. Although foam is composed of fluid and air mixture, some of the existing research also focus on generating foam particles, usually in a scale smaller than the fluid particles to be able to enhance the flow detail [TFK+03, GLR+06, LTKF08,

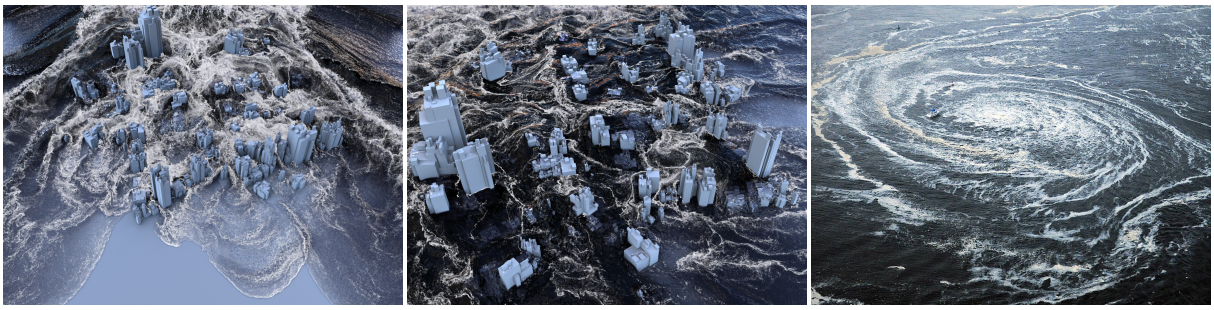


Figure 1: A flood scenario. Foam is rendered using our technique and composited with the rest of the scene (left and middle). Picture of real sea foam caused by a whirlpool (right) (©Reuters).

MMS09, IAAT12]. For foam generation, we employ [IAAT12]. The approach generates and processes three types of diffuse material, i.e. air bubbles, surface foam and spray. All types of diffuse material are represented with particles that are generated, advected and dissipated accoding to physically-motivated rules. The approach adds diffuse material to particle-based fluid simulations in a post-processing step.

For high quality foam renderings, ray-tracing methods are commonly preferred both for the fluid and the foam [GLR+06]. Although the fluid surface can be rendered efficiently using ray-tracing, non-homogenous phenomena such as foam require expensive volume rendering techniques. In [IAAT12], the authors employed a volume ray-casting method which accounts for absorption and emission of radiance but neglecting light scattering effects. In that method, each traced ray is sampled using equally spaced intervals; and according to the measured foam density at each sample point, the computed radiance is attenuated. The employed ray-casting approach, however, is time consuming to compute, especially for scenes with many millions of foam particles. The performance of volume ray-casting can be significantly improved by using the GPU-based method explained in [FAW10].

In [vdLGS09, BSW10], alternative to generating new particles, selected fluid particles are visualized as foam particles using GPU-based techniques for real-time applications. In [BSW10], Weber number thresholding is used to separate fluid and foam. Furthermore, the method also takes volumetric effects into account by rendering foam and fluid layers from back to front order. Therefore, it can visualize effects such as foam inside the fluid. Furthermore, based on the thickness of the foam, it generates foam color between two user defined colors. The approach, however, neglects information such as occlusion and irradiance from the environment when rendering foam, which limits its applicability to non-photorealistic real-time renderings.

There also exist methods for the modeling of larger scale foam effects by using air bubbles (e.g. see [KVG02, KLL+07, HLYK08, IBAT11, BDWR12]). In these works, air phase is either visualized by rendering spheres [KVG02, BDWR12], or by reconstructing the surface of the modeled air phase [KLL+07, HLYK08, IBAT11]. Since we are focusing on large scale scenarios, where the single air bubbles inside the foam are not clearly noticeable, such methods are beyond the scope of our paper.

1.3 Contribution

We present an efficient method for large scale foam rendering. In our approach, foam is rendered using a novel multi-pass rendering algorithm and finally composited with the pre-rendered images of the scene without foam. In comparison to volume ray-casting methods that compute only absorption and emission of radiance (e.g. [FAW10, IAAT12]), our approach is significantly faster as the foam particles are directly rendered. Furthermore, when compared to [BSW10], our pipeline takes the scene occlusion and lighting into account and therefore produces more convincing results that can be composited with realistic renderings. Results show that our new pipeline generates convincing large scale foam renderings (e.g. see Fig. 1) using modern GPU-based rendering architectures.

2 SCREEN SPACE FOAM RENDERING PIPELINE

As more air bubble layers implies more light scattering, we relate the foam thickness to the foam intensity as usually done in volume ray-casting. Later, we determine the regions on screen space which should receive, and therefore scatter less light using ambient occlusion and attenuate the foam intensity according to the occlusion factor. Afterwards, we approximate per-pixel foam irradiance to colorize the foam color according to the environment. Finally, the generated results are composited with the rest of the scene. We realized our approach using a seven pass rendering algorithm. The technical steps of our pipeline (illustrated in Fig. 2 and 3) can be summarized as:

- *PASS #1 and #2:* Storing eye space depth images of solid and fluid meshes in two textures, which are used to compute occlusion of foam fragments by those primitives in the later stages.

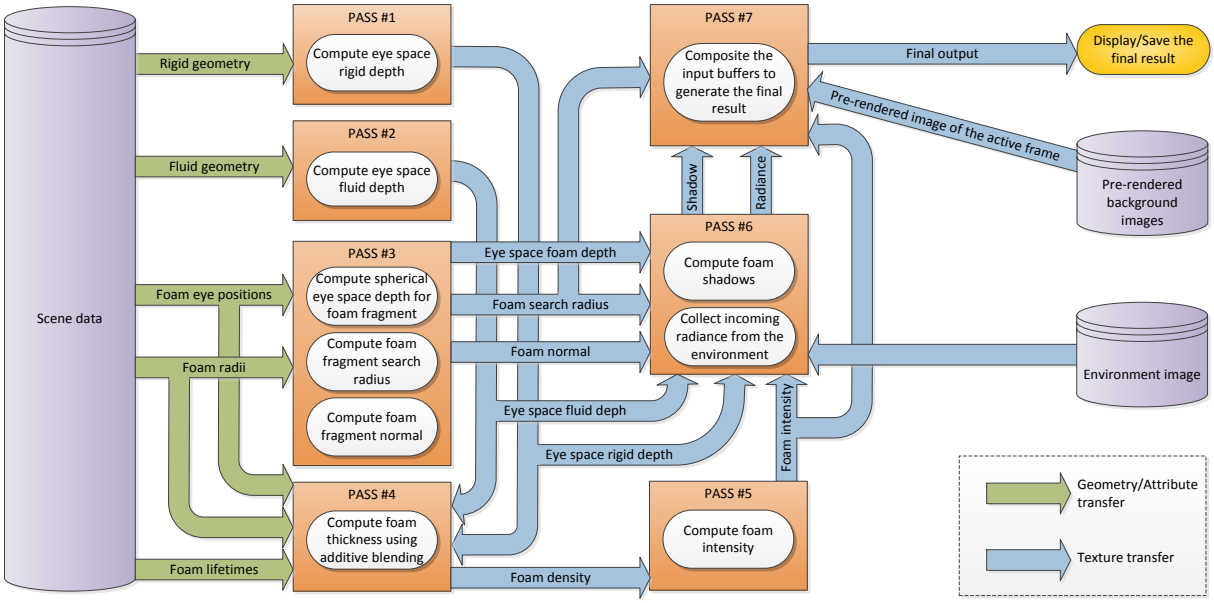
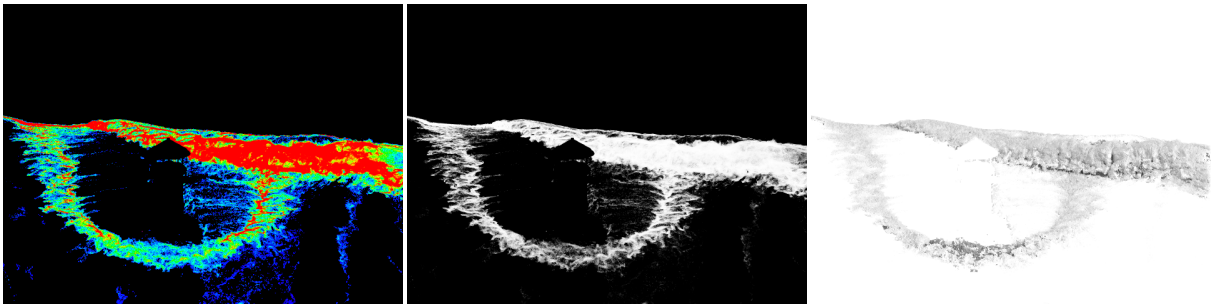


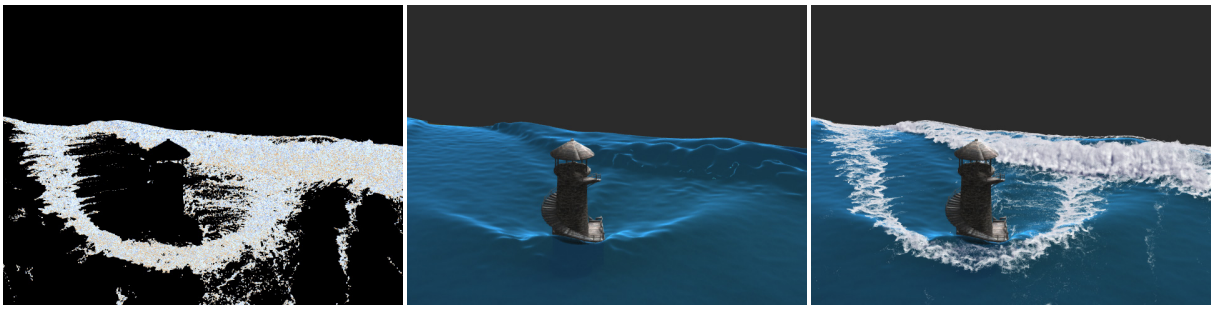
Figure 2: Diagram of our foam composition pipeline. Orange boxes denote the render passes and the arrows in between denote data flow and dependencies. For each frame, the render passes from #1 to #7 are executed. Each pass produces data explained in the enclosed rounded rectangles, which is then transferred through arrows to the subsequent passes. All of the generated textures have the same resolution as the final output.



(a) Foam thickness (red denotes thickest, blue denotes thinnest parts)

(b) Foam intensity

(c) Foam shadow (inverted)



(d) Foam irradiance

(e) Pre-rendered image

(f) Image composited with foam

Figure 3: Some of the intermediate textures from our foam composition pipeline (a-e) and the final composited result (f).

- *PASS #3:* Storing an eye space depth image of the foam particles in a texture, which is used in different parts of our pipeline. This pass also stores a search radius for each foam fragment, in whose range neighboring fragments are later considered for screen space ambient occlusion and final composition (Sec. 2.1). Additionally, this pass computes a normal for each foam fragment, which is used when approximating irradiance at the fragment location.
- *PASS #4:* Accumulating foam particles via additive blending to approximate per-pixel foam thickness. This pass also discards foam fragments that are occluded by solids and attenuates foam fragments that are inside of the fluid based on the fluid transparency (Sec. 2.2).
- *PASS #5:* Conversion of per-pixel foam thickness to per-pixel foam intensity (Sec. 2.3).
- *PASS #6:* Determination of foam fragments that should receive and scatter less light using screen space ambient occlusion (SSAO) and shadow generation for such regions (Sec. 2.4.1). This pass also approximates the irradiance at each foam fragment from an environment texture if the scene is illuminated using image based lighting (Sec. 2.4.2).
- *PASS #7:* Post processing of the foam and final composition with a pre-rendered image of the scene (Sec. 2.5).

Since the first step of the pipeline is relatively straightforward, we will focus on the remaining steps throughout this section. The following render passes are implemented using OpenGL Shading Language (GLSL).

2.1 Smoothed Depth and Search Radius Computation

We use point sprites instead of spheres for rendering foam particles. A regular point sprite has the same depth values for all of its fragments. However, to produce convincing results in the later steps of our pipeline, we modify the fragment depth values similar to [vdLGS09, BSW10], such that the spherical shapes of the particles are regained.

To create the initial depth information, foam particles with ids i and radii r_i in world space are rendered with depth testing and depth masking enabled. In [IAAT12], foam particles are separated to three different types, namely: spray, surface-foam and bubble particles. For bubble particles, we use half of r_i to make them less visible. Furthermore, particle radii are randomized as $r_i = \frac{r_i}{(i \bmod 5)+1}$ to make the particles look irregular between the scales $r_i/5$ and r_i.

The vertex shader computes eye space and projection space coordinates of the sprites and passes the resultant data to the fragment shader for further processing. In the fragment shader, the distance of the fragment position to the point sprite center is calculated using the sprite's texture coordinates to discard fragments that are outside of the circle. Afterwards, the flat depth values of the point sprite are transformed to spherical depth values. In this context, the first step is solving for the w coordinate of a unit sphere for the fragment's texture coordinates in uvw space as $w = \sqrt{1-u^2-v^2}$, where u and v denote texture coordinates of the fragment. Subsequently, the eye space z coordinate of the fragment is simply modified as

$$\mathbf{e}^{frag}_{foam_z} = \mathbf{e}^{frag}_{foam_z} + w \cdot r_i.$$

In contrast to [vdLGS09, BSW10], we do not apply filtering to the generated depth values since it would reduce the effect of ambient occlusion.

In the same render pass, the vertex shader also projects the search radius h_i for each particle as

$$h_i = \frac{r_i}{\tan\left(\frac{\alpha}{2}\right)\left|\mathbf{e}^{vert}_{foam_z}\right|},$$

where α is the field of view of the camera and $\mathbf{e}^{vert}_{foam_z}$ denotes z coordinate of the eye position of the point sprite (i.e., distance of the sprite to the camera). Afterwards, the search radius is passed to the fragment shader as h^{frag} to be written to a texture. The depth information and the search radius are essential when rendering the SSAO pass and when doing the final composition.

This pass also computes a world space normal for each fragment $\mathbf{n}_{frag}$ by transforming (u,v,w) using the transpose of the normal matrix, and stores the normals in a texture. Per fragment normals will be required when estimating irradiance in Sec. 2.4.2.

2.2 Thickness Estimation

Before estimating the intensity of foam at a given pixel position, we estimate the foam thickness for each pixel. In this step, foam particles are rendered again as point sprites with the spherical depth modification as in the previous render pass. Similar to [vdLGS09, BSW10], the foam fragments are blended additively to estimate thickness. Different from [vdLGS09, BSW10], however, depth buffer read and write is disabled as we do not require the frontmost particles to be visible.

As foam particles are separated to spray, surface-foam and bubble particles, we also employ this knowledge to render foam fragments differently by using a falloff function with different arguments, where the falloff is based on the fragment's distance to the particle center in texture coordinates. The falloff function f is defined as

$$f(x,b,n,m) = \begin{cases} \left[1-\left(\frac{x}{b}\right)^n\right]^m & \frac{x}{b} \le 1 \\ 0 & \text{otherwise} \end{cases}, \quad (1)$$

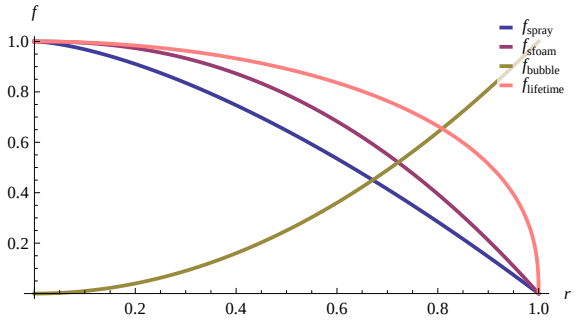


Figure 4: Different forms of the falloff function given in (1) that are used in our experiments .

where x is the distance to the center, b is the maximum allowed distance, and $n \geq 0$ and $m \geq 0$ are exponents which determine the shape of the function (e.g., $n = 1$ and $m = 1$ result in linear falloff). When rendering spray, surface-foam and bubble fragments, we used $f_{spray} = f(x, 1, 1.5, 1)$, $f_{sfoam} = f(x, 1, 2.25, 1)$ and $f_{bubble} = 1 - f(x, 1, 2, 1)$ respectively. These different falloff functions are illustrated in Fig. 4 and the corresponding particle intensities are shown in Fig. 5. We preferred a larger overall intensity for surface foam particles to increase their visibility. Whereas, we preferred a comparatively smaller intensity value for the spray particles to make them relatively less visible. Furthermore, we used hollow circle like structures for the bubble particles to make their appearance more convincing under water.

In this step, the intensities of the foam particles are further modulated based on two additional factors. The first of these factors is the lifetime of the particle. For this purpose, we use $f_{lifetime} = f(l_i, 1, 2, 0.4)$, where $0 < l_i < 1$ denotes the normalized lifetime of a particle. Such a function allows a foam particle to remain visible for a sufficiently long time and fade smoothly near the end of its lifetime. Furthermore, when a particle lies in the back of the closest fluid surface (i.e. $0 < \mathbf{e}^{frag}_{fluid_z} < \mathbf{e}^{frag}_{foam_z}$, where $\mathbf{e}^{frag}_{fluid_z}$ is the eye space z coordinate of the fluid surface), we apply an additional falloff to its intensity, which is defined as

$$f_{att} = f(\mathbf{e}^{frag}_{foam_z} - \mathbf{e}^{frag}_{fluid_z}, \eta_{max}, \eta_n, \eta_m),$$

with the limiting distance η_{max}, where the foam fragment completely fades to invisible, and η_n and η_m are the exponents for shaping the attenuation curve.

At the end of this render pass, the final foam thickness values are stored in a texture (see Fig. 3a). In the next pass, the computed thickness values are processed and converted to normalized intensity values to lie between 0 and 1. For all subsequent passes, a screen-filling quad is rendered to further process the relevant information that are saved in the textures.

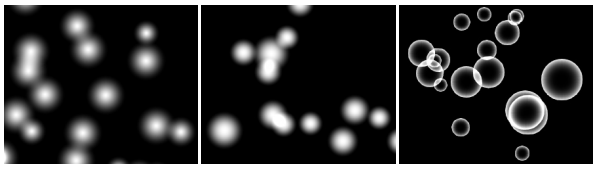


Figure 5: Intensity distributions of different types of foam particles, namely: spray particles (left), surface foam particles (middle) and air bubble particles (right).

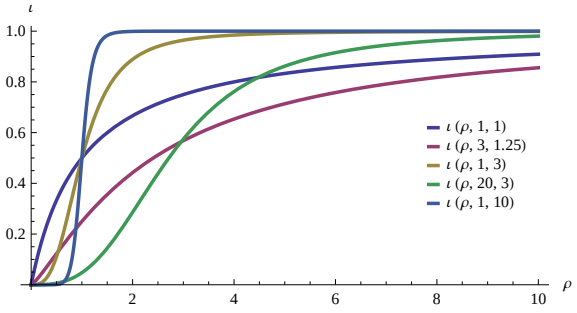


Figure 6: Different forms of the sigmoid function that can be applied to the accumulated foam densities. The function can be used to create different distributions as well. For instance, to reduce the intensities below some threshold, $\rho_{exp} \geq 2$, can be used. We use the $\iota(\rho, 3, 1.25)$ form in our experiments.

2.3 Intensity Estimation

As foam is composed of more bubble layers, it scatters more of the incoming light. We use this knowledge to relate the foam intensity proportional to foam thickness. A texel from the previous render pass may have any value between $[0, \infty)$. In this render pass, we scale the values taken from that texture to the interval $[0, 1]$. However, scaling the values linearly to the target interval would make sparse areas invisible. We expect the foam to become completely opaque after some thickness threshold. Therefore, to increase the effective range of the thinner regions, to reduce the range of thicker regions and to normalize the intensities, we define the following sigmoid function ι to non-linearly scale a pixel thickness value ρ as

$$\iota(\rho, \rho_{mod}, \rho_{exp}) = \frac{\rho^{\rho_{exp}}}{\rho_{mod} + \rho^{\rho_{exp}}},$$

where $\rho_{mod} > 0$ and $\rho_{exp} > 0$ control how fast the function grows. Note that if $\rho > 0$ and $\rho_{exp} > 0$, $0 < \iota < 1$. ι is illustrated in Fig. 6 for different parameters. Furthermore, Fig. 7-top shows the effect of using different ρ_{mod} values.

At the end of this step, the normalized intensities are saved in a texture, which will be used in the following steps (see Fig. 3b).

Figure 7: Application of different parameters for the setting presented in Fig. 1-middle. top-left: $\rho_{mod} = 1$; top-right: $\rho_{mod} = 5$; bottom-left: $AO_{ShScale}$ =0.1; bottom-right: $AO_{ShScale} = 2$.

2.4 Foam Radiance Estimation

Since foam is composed of many transparent layers of air bubbles, light can travel through it and then scatter. Until the current stage of our pipeline, we assume that foam scatters light uniformly, where the intensity of the light was only related to the foam thickness. In this section, we determine the regions which should receive, and therefore scatter less light using ambient occlusion (AO), and generate shadows for these regions (Sec. 2.4.1). Furthermore, the intensities that are computed in the previous section do not employ any knowledge about the actual illumination that comes from the scene. In this render pass, we will also use a very rough screen space approximation of the irradiance from the surrounding environment, which is used to colorize the foam fragments (Sec. 2.4.2).

This render pass again gets the textures that have been computed in the previous step as input and computes two additional textures, one for the shadow and another for the illumination of the foam (see Fig. 3).

2.4.1 Shadow Generation

As object space AO methods (e.g. [ZIK98, Bun05, RWS+06]) are very expensive to compute, especially for complex dynamical phenomena such as foam, we investigated SSAO techniques [TCM06, Mit07, SA07, RGS09, BS09, HL10]. Finally, we decided to build our SSAO approach upon the basic concept explained in [Mit07] because of its efficiency and simplicity. One important difference of our method in comparison to [Mit07] is that we apply multiple sample collection iterations to capture both small scale and large scale occlusions. Instead of increasing search radii, [HL10] used multiple depth maps with decreasing resolution to achieve the same effect. The search radii and total number of passes are controlled by three parameters: the initial search radius factor $AO_{InitSRFac}$, which is a factor for h^{frag} to capture small scale occlusions; the search radius increment factor $AO_{SRIncFac}$, which is another factor for h^{frag} to determine how much the search radius increases in each sample collection step; and finally $AO_{\#Passes}$, which limits the total number of SSAO passes. For each fragment, 3d samples are generated within the fragment search radius:

$$h^{frag}_{pass} = h^{frag}\left(AO_{InitSRFac} + AO_{SRIncFac} \cdot AO_{Pass}\right),$$

where AO_{pass} increases by 1 in each sample collection pass and $AO_{pass} \leq AO_{\#Passes}$. In our experiments we used: $AO_{InitSRFac} = 1$, $AO_{SRIncFac} = 7$ and $AO_{\#Passes} = 3$.

The total number of samples ν in each sample collection pass is controlled by a user defined sampling density parameter AO_{SDens} as

$$\nu = \text{clamp}\left(\frac{3}{4}\pi {h^{screen}_{pass}}^3 AO_{SDens}, AO_{\#MinSamp}, AO_{\#MaxSamp}\right),$$

where h^{screen}_{pass} is the search radius projected to fragment coordinates. Since h^{frag} can be very small for distant fragments, a minimum value $AO_{\#MinSamp}$ is used for ν. An upper limit $AO_{\#MaxSamp}$ is also introduced to prevent too many samples from being generated for fragments that are very close to the viewer. In our experiments, we used $AO_{SDensity} = 0.5$, $AO_{\#MinSamp} = 16$ and $AO_{\#MaxSamp} = 512$. The samples are created inside a cube in the range $[-1,1]$ on all axes using the Halton sampling algorithm with a constant seed [Hal64], which produces low-discrepancy sequences. Subsequently, the samples are mapped to a sphere by simply neglecting the samples that lie outside of the sphere in the range $[-1,1]$.

Additionally, the occlusion contribution λ of a sample $\mathbf{s}$ depends on its distance to the fragment and we compute it using a quadratic falloff as

$$\lambda = (1 - |\mathbf{s}|)^2.$$

Furthermore, if a sample is occluded by a fragment with a distance larger than the user defined $AO_{MaxOcclDist}$, the sample does not contribute to the visibility of the fragment. This effect is necessary to prevent occlusion by distant fragments and is controlled using a quadratic falloff function as

$$\delta = \max\left[\left(1 - \frac{\left|\mathbf{e}^{frag}_{foam_z} - s_z\right|}{AO_{MaxOcclDist}}\right), 0\right]^2,$$

where $AO_{MaxOcclDist} = 5$ is used in our experiments. The sample $\mathbf{s}$ is used to look up the occlusion in eye

space by other fragments (e.g. foam, fluid and solid fragments) in the scene. Based on the knowledge collected so far, the occlusion k of a sample is defined as

$$k = \begin{cases} 1 & \left[\left(s_z > \mathbf{e}^{frag}_{foam_z} \vee s_z > \mathbf{e}^{frag}_{fluid_z} \vee \right.\right. \\ & \left.\left. s_z > \mathbf{e}^{frag}_{rigid_z}\right) \wedge (0 < \delta < 1)\right] \\ 0 & \text{otherwise} \end{cases},$$

which basically states that; if a sample is occluded by any other fragment in the scene and if the occlusion distance is not larger than $AO_{MaxOcclDist}$, the sample is occluded.

Afterwards, we compute the occlusion factor ω of a fragment as

$$\omega = \frac{\sum_{AO_{pass}=1}^{AO_{\#Passes}} \left(\sum_{i=1}^{\psi} \lambda_i \cdot \delta_i \cdot k_i \cdot a_i\right)}{\sum_{AO_{pass}=1}^{AO_{\#Passes}} \left(\sum_{i=1}^{\psi} \lambda_i\right)},$$

where for the pass AO_{pass}, i iterates over all generated samples that are inside the render area (denoted as ψ), and a_i is the transparency of the sampled fragment, which is equivalent to ι_i for foam fragments. For rigid and fluid fragments, a_i is equivalent to the fragment's transparency. Additionally, if there are multiple overlapping transparent fragments at a sample position, a_i is computed by adding all of the transparency values.

Finally, so as to be more flexible about the appearance of the generated shadows, we compute the final shadow value ζ clamped into $[0,1]$ as

$$\zeta = \text{clamp}\left[\left(\omega \cdot AO_{ShScale}\right)^{AO_{ShExp}} + AO_{ShOffset}, 0, 1\right]$$

which is controlled by three self-explanatory user defined parameters. In the presented scenarios, we used: $AO_{ShScale} = 1$, $AO_{ShExp} = 1.5$ and $AO_{ShOffset} = -0.05$. The ambient occlusion step especially improves the regions that have similar intensities, which would look totally flat otherwise (e.g., see Fig. 8, top-middle). Furthermore, Fig. 7-bottom shows the effect of different $AO_{ShScale}$ values. The computed ζ values are written to a texture to be further used by the final composition step (see Fig. 3c).

2.4.2 Irradiance

If the scene is illuminated using image based lighting, we approximate the direct illumination of each foam fragment by looking up the environment map that has been used as the light source. Using the fragment normal $\mathbf{n}^{frag}$, we create a hemisphere around the normal and use the already generated samples from the SSAO step to create direction vectors $\mathbf{n}_i^{sample}$ that are used for looking up the intensity $\mathbf{P} = (r,g,b)$ at an environment map position. Finally, the irradiance that is coming from the environment to a fragment location is simply computed in a cosine weighted fashion as

$$\mathbf{I} = \left(\frac{\sum_{i=1}^{\psi} \mathbf{P} \cdot \left(\mathbf{n}_i^{sample} \cdot \mathbf{n}^{frag}\right)}{\psi}\right),$$

where i iterates only over the samples that are generated for the first sample collection pass. The sole purpose of this step is to reflect the hue of the environment onto the foam fragments to make the foam not look too distinct from the rest of the scene. Finally, the computed $\mathbf{I}$ values are written to another texture to be used by the next and the final render pass (see Fig. 3d). The performance of this step can be improved by using an irradiance environment map and making color look-up once for every $\mathbf{n}^{frag}$.

2.5 Composition

In this render pass, the information that has been created in the previous steps and the pre-rendered images of the scene without foam are composited to generate a final image of the scene with foam (see Fig. 2).

Depending on the user defined $AO_{\#MaxSamples}$, the shadow and radiance values computed in the previous section can include high frequency noise. In order to alleviate this problem, we apply Gaussian blur with a filter radius of $\frac{3}{2}h^{screen}_{pass}$ to both textures to generate per-pixel $\zeta_{filtered}$ and $\mathbf{I}_{filtered}$ before doing the composition.

Afterwards, to compute a final shadow color ζ_{final} for a pixel, the filtered shadow values are modulated with a user defined color $\mathbf{C}_{ShadowColor}$ and clamped into $[0,1]$ as

$$\boldsymbol{\zeta}_{final} = \text{clamp}\left[\zeta_{filtered} \cdot (\mathbf{C}_{white} - \mathbf{C}_{ShadowColor}), 0, 1\right],$$

where $\mathbf{C}_{white} = (1,1,1)$. We select $\mathbf{C}_{ShadowColor}$ similar to the visible color of the fluid that the foam is generated on, and it was chosen in our experiments as $(0,0,0.2)$ because of the dark blue appearance of the fluids in our renderings. Since $\boldsymbol{\zeta}_{final}$ will be subtracted when doing the composition, the $\mathbf{C}_{ShadowColor}$ term is subtracted from white to invert it. From our experiences, colorizing shadows makes the foam blend better with the underlying fluid.

As foam is composed of many air-liquid interfaces, it has a very large scattering albedo which causes it to scatter most of the incoming light, but absorb only a small amount of it. Therefore, it is usually observed very bright. We control this phenomenon by linearly scaling the irradiance values $\mathbf{I}$ using a user defined parameter $C_{IrrScale}$, whose value depends on the desired foam brightness and the color range of the environment map used. Afterwards, we clamp the resulting color into the $[0,1]$ interval to compute

$$\mathbf{I}_{final} = \text{clamp}\left(C_{IrrScale} \cdot \mathbf{I}_{filtered}, 0, 1\right).$$

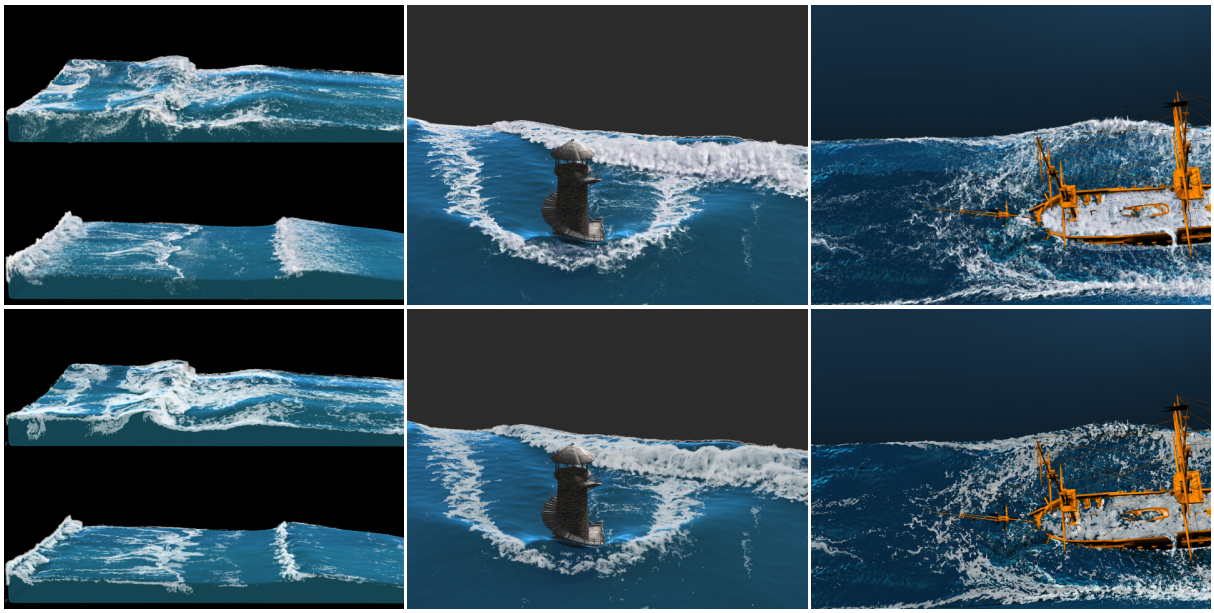


Figure 8: Comparisons of our method (top) to volume ray-casting that computes emission and absorption only (bottom). As our method approximates shadows in concave regions, the foam looks more volumetric and detailed. The scenes are named from left to right as: Wave, Lighthouse and Ship.

	# Foam particles	Resolution	Average foam rendering time per frame			
			Ray-casting [IAAT12]	Ray-casting [FAW10]	Ours (intensity only)	Ours (total)
Wave	up to 820K	800×600	2 min 10 s	235 ms	8 ms	52 ms
Ship	up to 9M	800×600	4 min 20 s	760 ms	16 ms	102 ms
Lighthouse	up to 15M	800×600	7 min 3 s	1 s	21 ms	150 ms
Flood	up to 29M	1280×960	16 min 19 s	1.7 s	33 ms	235 ms

Table 1: Performance analysis of the example scenes.

Finally, the composited pixel color $\mathbf{C}$ is computed as

$$\mathbf{C} = (1-\iota)\,\mathbf{C}_{bg} + \iota\left(\mathbf{I}_{final} - \boldsymbol{\zeta}_{final}\right)$$

where $\mathbf{C}_{bg}$ is the color at the corresponding pixel position of the background image on which the foam is composited (see Fig. 3f).

3 RESULTS

In this section, we demonstrate the versatility of our approach in different animation sequences. For all presented scenes, the underlying fluid has been simulated using the methods referred in [IAAT12], and the fluid surfaces have been reconstructed based on [SSP07, AIAT12, AAIT12]. The scenes were rendered using mental ray [NVI11] on an Intel Xeon X5690 CPU with 12 GB RAM, and the foam composition pipeline was implemented using GLSL and ran on an NVIDIA 480 GTX GPU with 1.5 GB RAM. The ray-casting code used in [IAAT12] was implemented as a mental ray shader and ran on the CPU, and an optimized version based on [FAW10] was implemented on the GPU. All scenes were illuminated using image based lighting with a clear sky environment map.

For all scenes, foam was simulated using [IAAT12] and the same foam data were used for the rendering comparisons to [IAAT12]. For the comparisons shown in Fig. 8, the ray-casting technique explained in [IAAT12] took 9 s to 20 min depending on the complexity of the frame, excluding the other scene geometry and loading of the foam data. Using the optimized volume ray-casting scheme, the computation time has been reduced down to 90 ms to 2.5 s. Using our pipeline, the foam rendering of a frame took 30 ms to 270 ms depending on the complexity of the foam in the scene being rendered, excluding the time spent for loading of the foam data from secondary storage to the GPU memory. The results produced by using a basic volume ray-casting scheme that only accounts for absorption and emission of radiance is similar to the results we achieve excluding the additional effects that are described in Sec. 2.4 (see also Fig. 3b). Excluding those additional effects, our pipeline took between 5 ms to 39 ms per frame. See Table 1 for additional information about each scene. As our pipeline also takes additional effects into account (i.e. ambient occlusion and irradiance estimation), our presented foam renderings look volumetric and blend with the rest of the scene (see Fig. 8). Note that in [IAAT12], the fluid surface has been constructed only

for the fluid particles that have more than five neighbors. For our comparisons to [IAAT12], however, we used the whole fluid surface for our renderings to better estimate the SSAO of the foam by the fluid surface. Therefore, differences between the two fluid surfaces can be noticeable.

For all of our scenes, most of the rendering time has been spent on the foam radiance estimation pass (between 50-80%). Whereas, the computational overheads of the rest of the render passes were significantly lower.

4 DISCUSSION AND FUTURE WORK

Taking a closer look at sea foam from a distance less than a few meters, one may observe the underlying air bubbles at varying sizes which form the foam. Rendering of such scenarios is not handled by our approach. However, using an air bubble generation technique like [BDWR12] for such close-ups might be an interesting direction for future research.

For scenes where most of the light is coming from a specific direction at shallow angles (e.g. sunset scenarios), the currently employed SSAO based shadow generation technique can fail to capture the resultant potentially large shadows cast by distant objects. For such cases, an explicit shadow generation algorithm which can handle image based lighting such as the one explained in [CK09], or explicit shadow source selection as discussed in [Bjo04] can be employed. Since we assume that foam scatters most of the incident light randomly, we omitted Fresnel effect. However, it might be desirable to make the foam reflect the environment, when it is viewed from a shallow angle.

Our algorithm neglects many physical effects that could be otherwise simulated by using modern ray-tracing techniques. Those effects include; scattering of light inside the foam, influence of the foam on the appearance of the surrounding objects and vice versa. However, for large scale scenarios (e.g. as in Fig 1), those effects have less significance on the appearance of the foam, and our approximations can efficiently generate convincing results. However, for close-ups, the effects that we have omitted have more significance on the final outcome. For those cases, using a volume ray-casting method that simulates light scattering can definitely yield more convincing results (e.g. [RNGF03, GLR$^+$06]).

Although we demonstrated our rendering scheme only for the particle data generated by the method explained in [IAAT12], we believe that our pipeline is mostly applicable to the rendering of other particle based foam simulation techniques.

5 CONCLUSION

We presented an efficient, screen-space foam rendering pipeline that can render large particle-based foam data sets on the GPU. Our approach uses a multi-pass rendering scheme, where different effects are added to the foam rendering incrementally, and the final foam rendering is composited with a pre-rendered image of the scene. The presented method can be used as an efficient alternative to ray-casting techniques for the rendering of large scale particle-based foam data.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments. This project is supported by the German Research Foundation (DFG) under contract numbers SFB/TR-8 and TE 632/1- 2. We also thank NVIDIA ARC GmbH for supporting this work.

6 REFERENCES

[AAIT12] G. Akinci, N. Akinci, M. Ihmsen, and M. Teschner. An efficient surface reconstruction pipeline for particle-based fluids. In *Workshop on Virtual Reality Interaction and Physical Simulation*, pages 61–68. The Eurographics Association, 2012.

[AIAT12] Gizem Akinci, Markus Ihmsen, Nadir Akinci, and Matthias Teschner. Parallel surface reconstruction for particle-based fluids. *Computer Graphics Forum*, 31:1797–1809, 2012.

[BDWR12] O. Busaryev, T.K. Dey, H. Wang, and Z. Ren. Animating bubble interactions in a liquid foam. *ACM Trans. on Graphics (SIGGRAPH Proc.)*, 31(4):63, 2012.

[Bjo04] K. Bjorke. *Image-based lighting*. GPU Gems. NVIDIA, 2004.

[BS09] L. Bavoil and M. Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, page 45. ACM, 2009.

[BSK$^+$07] R. Bredow, D. Schaub, D. Kramer, M. Hausman, D. Dimian, and R.S. Duguid. Surf's up: the making of an animated documentary. In *ACM SIGGRAPH 2007 courses*, volume 5, 2007.

[BSW10] F. Bagar, D. Scherzer, and M. Wimmer. A layered particle-based fluid model for real-time rendering of water. In *Computer Graphics Forum*, volume 29, pages 1383–1389. Wiley Online Library, 2010.

[Bun05] M. Bunnell. Dynamic ambient occlusion and indirect lighting. *Gpu gems*, 2(2):223–233, 2005.

[CK09] Mark Colbert and Jaroslav Krivanek. Real-time dynamic shadows for image-based lighting. *ShaderX 7 - Advanced Rendering Techniques*, page Section 4.3, 2009.

[CM10] N. Chentanez and M. Müller. Real-time simulation of large bodies of water with small scale details. In *Proc. of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 197–206, 2010.

[FAW10] R. Fraedrich, S. Auer, and R. Westermann. Efficient high-quality volume rendering of sph data. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1533–1540, 2010.

[GLR+06] W. Geiger, M. Leo, N. Rasmussen, F. Losasso, and R. Fedkiw. So real it'll make you wet. In *ACM SIGGRAPH 2006 Sketches*, 2006.

[Hal64] J.H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964.

[HL10] T.D. Hoang and K.L. Low. Multi-resolution screen-space ambient occlusion. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, pages 101–102. ACM, 2010.

[HLYK08] Jeong-Mo Hong, Ho-Young Lee, Jong-Chul Yoon, and Chang-Hun Kim. Bubbles alive. *ACM Trans. on Graphics (SIGGRAPH Proc.)*, 27:48:1–48:4, 2008.

[hyb11] Next Limit Technologies: Realflow 2012, Hybrido. White Paper. 2011.

[IAAT12] Markus Ihmsen, Nadir Akinci, Gizem Akinci, and Matthias Teschner. Unified spray, foam and air bubbles for particle-based fluids. *The Visual Computer*, pages 1–9, 2012. 10.1007/s00371-012-0697-9.

[IBAT11] M. Ihmsen, J. Bader, G. Akinci, and M. Teschner. Animation of air bubbles with SPH. In *International Conference on Graphics Theory and Application*, pages 225–234, 2011.

[KLL+07] Byungmoon Kim, Yingjie Liu, Ignacio Llamas, Xiangmin Jiao, and Jarek Rossignac. Simulation of bubbles in foam with the volume control method. *ACM Trans. on Graphics (SIGGRAPH Proc.)*, 26(3), July 2007.

[KVG02] Hendrik Kück, Christian Vogelgsang, and Günther Greiner. Simulation and rendering of liquid foams. In *In Proc. Graphics Interface '02*, pages 81–88, 2002.

[LTKF08] F. Losasso, J. Talton, N. Kwatra, and R. Fedkiw. Two-way coupled SPH and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, 2008.

[Mit07] M. Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*. ACM, 2007.

[MMS09] V. Mihalef, D. Metaxas, and M. Sussman. Simulation of two-phase flow with sub-scale droplet and bubble effects. In *Computer Graphics Forum*, volume 28, pages 229–238. Wiley Online Library, 2009.

[MSD07] M. Müller, S. Schirm, and S. Duthaler. Screen Space Meshes. In *Proceedings of ACM SIGGRAPH / EUROGRAPHICS Symposium on Computer Animation (SCA)*, 2007.

[NVI11] NVIDIA ARC. mental ray 3.9 [software]. http://www.mentalimages.com/products/mental-ray/about-mental-ray.html, 2011.

[RGS09] T. Ritschel, T. Grosch, and H.P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 75–82. ACM, 2009.

[RNGF03] N. Rasmussen, D.Q. Nguyen, W. Geiger, and R. Fedkiw. Smoke simulation for large scale phenomena. *ACM Trans. on Graphics (SIGGRAPH Proc.)*, 22(3):703–707, 2003.

[RWS+06] Z. Ren, R. Wang, J. Snyder, K. Zhou, X. Liu, B. Sun, P.P. Sloan, H. Bao, Q. Peng, and B. Guo. Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation. In *ACM Trans. on Graphics (SIGGRAPH Proc.)*, volume 25, pages 977–986. ACM, 2006.

[SA07] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 73–80. ACM, 2007.

[SSP07] B. Solenthaler, J. Schläfli, and R. Pajarola. A unified particle model for fluid-solid interactions. *Computer Animation and Virtual Worlds*, 18(1):69–82, 2007.

[TCM06] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1237–1244, 2006.

[TFK+03] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki. Realistic Animation of Fluid with Splash and Foam. *Computer Graphics Forum*, 22(3):391–400, 2003.

[vdLGS09] W.J. van der Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98. ACM, 2009.

[YT10] J. Yu and G. Turk. Reconstructing surfaces of particle-based fluids using anisotropic kernels. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 217–225. Eurographics Association, 2010.

[ZB05] Y. Zhu and R. Bridson. Animating sand as a fluid. In *ACM Trans. on Graphics (SIGGRAPH Proc.)*, pages 965–972, New York, NY, USA, 2005. ACM Press.

[ZIK98] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. *Rendering techniques*, 98:45–55, 1998.

VDR-AM: View-Dependent Representation of Articulated Models

Pio Claudio
Dept. of CS, KAIST
pioclaudio@gmail.com

Dohyeong Kim
Purdue University
karkayan@gmail.com

Tae-Joon Kim
Dept. of CS, KAIST
tjkim.kaist@gmail.com

Sung-eui Yoon
Dept. of CS, KAIST
sungeui@gmail.com

ABSTRACT

We present a novel, View-Dependent Representation of Articulated Models (VDR-AM), and show its main benefits in the context of view-dependent rendering integrated with occlusion culling for large-scale crowd scenes. In order to provide varying resolutions on each animated, articulated model, we propose to use a cluster hierarchy in the VDR-AM for an articulated model. The cluster hierarchy serves as a dual representation for both view-dependent rendering and occlusion culling. For a high-performance view-dependent rendering and occlusion culling, we construct each cluster of the cluster hierarchy to contain a spatially coherent portion of the mesh that also has similar simplification errors. To achieve our goal, we present an error-aware clustering method for articulated models. We also identify a subset of animation poses that well represents the original pose data and perform the well-known quadrics-based simplification to efficiently compute our representation, while achieving a high quality simplification. At runtime, we choose a LOD cut from the cluster hierarchy given a user specified error bound in the screen space and render all the visible clusters in the LOD cut. We implement our method in GPU and achieve interactive performance (e.g., 40 frames per second) for large-scale crowd scenes that consist up to thousands of articulated models and 242 M triangles, without noticeable visual artifacts.

Keywords

View Dependent Rendering, Character Animation, Level of Detail

1 INTRODUCTION

Owing to advances of data capture and modeling technologies, detailed articulated models are easily generated and widely used in many different applications. Moreover, various crowd simulation techniques have been designed and a high number of articulated models are frequently used in large-scale crowd scenes [TOY+07, NGCL09]. In typical crowd scenes with lots of articulated characters, it can require high computation costs of rendering and performing other operations (e.g., collision detection) for handling those articulated characters.

A significant amount of research has been put in order to improve the performance of rendering and conducting various operations for polygonal models. Some of them include designing multi-resolution representations of polygonal models [LRC+02, YGKM08], performing visibility culling [COCSD03], collision detection [TCYM09], etc. Unfortunately, most of these prior techniques assume polygonal models and do not easily apply to articulated models. This is mainly because articulated characters are dynamically animated with an underlying deformation model (e.g., skeleton) that changes the shapes of characters.

Figure 1: This figure shows two images of an exhibition crowd scene that has 1 K articulated characters. All the characters and the scene have 83 M triangles. Our method achieves 46 frames per second (fps) on average for this model with 0.5 pixel-of-error (PoE) in a 1280 by 720 HD screen resolution.

In terms of rendering articulated models, many techniques have been proposed to improve the rendering performance. At a high level, these techniques can be classified as image-based [DHOO05, KDC+08] and mesh simplification approaches [MG03, DR05, LS09] for articulated models. Image-based approaches have been reported to achieve a high rendering performance by rendering textures instead of triangles. However, these techniques may provide low quality rendering results for certain views (e.g., overhead or near views [DHOO05, KDC+08]) or may not be used

for other geometric applications such as collision detection.

Geometric simplification methods for articulated models, on the other hand, can provide high-quality polygonal meshes that can be used for various views. However, these simplification methods have focused on computing different versions of level-of-detail (LOD) meshes from an input articulated mesh, and have not been applied widely to view-dependent rendering that uses different LODs for portions of the mesh depending on viewing information. To the best of our knowledge, there have been no prior methods that adopt view-dependent rendering integrated with culling for large-scale scenes consisting of hundreds or thousands of articulated models [RD05].

Main contributions. In this paper we propose a novel, View-Dependent Representation of Articulated Models, VDR-AM. We show its benefits in the context of rendering, especially view-dependent rendering integrated with occlusion culling. VDR-AM consists of a cluster hierarchy that serves as both a multi-resolution representation and a bounding volume hierarchy. We use its multi-resolution representation for view-dependent rendering, and its bounding volume hierarchy for occlusion culling. We construct each cluster of the cluster hierarchy to contain a spatially-coherent small portion of the mesh that also has similar simplification errors. To construct such clusters, we propose an error-aware clustering method. Given the cluster hierarchy of VDR-AM, we can compute a LOD cut that satisfies a user-specified screen space error in terms of pixels-of-errors (PoE) at runtime. We also perform occlusion culling for clusters in the LOD cut in an efficient manner that utilizes the temporal coherence between consecutive frames.

We have implemented our method and applied it to three large-scale crowd scenes that consist of up to five thousand articulated models that have 242 M triangles in total. Even though our VDR-AM representation is not mainly designed for static polygonal models, it can be naturally applied to handling those models without major modifications. Therefore, we have also applied our representation even for static models that are parts of tested crowed scenes. Our method shows 16 to 50 frames per second (fps) without visible artifacts in a 1280 by 720 HD screen resolution. Compared with a base rendering method that uses the original articulated models combined with view-frustum culling, our method achieves four to eight times improvement.

2 RELATED WORK

In this section we give a background on animating articulated models based on skinning, and briefly review previous work related to our method. For detailed information about crowd rendering in general, refer to an excellent survey by Ryder et al. [RD05].

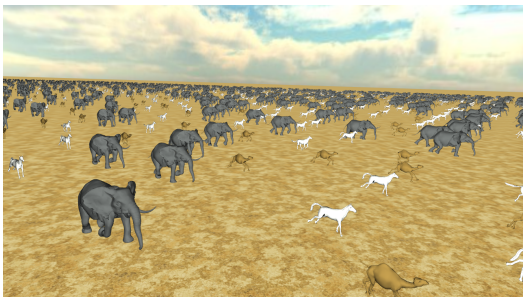


Figure 2: This figure shows a stampede crowd scene that has 5 K articulated characters and 242 M triangles. Our method can achieve 16 frames per second (fps) on average for this large-scale crowd scene.

2.1 Background of Articulated Models

Articulated models are typically designed with skinning and skeleton animations. In this case an articulated model consists of a base mesh, a skeleton, and vertex weights. The base mesh, also known as skin, is a 3D polygonal mesh, and the skeleton is a hierarchical representation of bones. The vertex weights associated with vertices of the base mesh represent the skin-to-skeleton binding. One of the most popular techniques used to achieve interactive animation is linear blend skinning.

The animation of the base mesh of an articulated model is defined by a series of poses. Each pose is defined by a 4 by 4 transformation matrix representing positions and orientations of bones of the skeleton. A vertex position, v, in the base mesh is then moved to a new position, $\hat{v}$, by linear blend skinning with a pose, based on the following equation:

$$\hat{v} = \sum_i w_i M_i v, \tag{1}$$

where M_i is the transformation of the ith bone associated with the vertex with the weight w_i given the input pose. There are more advanced skin blending methods such as skinning using the dual quaternions, and they can be also used with our method.

2.2 Mesh Simplification

Mesh simplification has been extensively studied for polygonal meshes, and numerous techniques have been presented. Among them the quadric error metric (QEM) and edge collapse operations [GH97] have been most widely used for high-quality mesh simplifications.

Many simplification techniques have been also proposed for dynamic models, but some of them [KG05, HCC06] are not directly applicable to articulated models that are animated with an underlying deformation model. In this section we focus on simplification methods that can handle articulated models with skinning.

Simplification for articulated models. Mohr and Gleicher [MG03] applied the QEM method for simplifying articulated meshes. Their method takes a skinned

mesh and a series of example poses. It computes the quadric error for each vertex by considering all the example poses. DeCoro and Rusinkiewicz [DR05] improved this method by considering the bone transformations and computing the quadric error in a base reference pose. Recently, Landreneau and Schaefer [LS09] perform the simplification by considering weights associated with vertices and thus achieve higher simplification quality over other techniques. Our simplification method is based on the work of DeCoro and Rusinkiewicz [DR05], but can be improved by adopting techniques proposed by Landreneau and Schaefer [LS09]. In addition, Pilgrim et al. [PSA07] proposed a progressive skinning technique that progressively uses a subset of original bones of articulated models. This technique can be combined with our method for higher rendering performance.

Image-based simplification methods. As an alternative representation for polygonal representations, image-based representations [DHOO05, KDC+08] like impostors have been proposed for articulated models and reported to achieve a high rendering performance, while maintaining a reasonable memory requirement [YYBE13]. As downsides, these techniques may provide low quality rendering results for certain views (e.g., overhead views) or require a high storage requirement. In addition, image-based representations may provide low-quality results for various geometric operations such as collision detection. Nonetheless, these image-based representations can be used together with polygonal representations including ours, to achieve a higher rendering performance and quality. For example, one can use polygonal representations in a near field and use imposters in a far field, as suggested by Kavan et al. [KDC+08]. As a result, our method is orthogonal to image-based techniques in the context of rendering.

2.3 View-Dependent Rendering and Culling

View-dependent rendering (VDR) uses lower resolutions for portions of the mesh that are located farther away from the viewer. This technique aims to reduce the number of rendered triangles of complex models depending on viewing configurations and has been extensively studied for polygonal meshes [YGKM08]. VDR originated as an extension of the progressive mesh (PM) that is a linear sequence of encoding finer meshes [Hop97]. Hoppe [Hop97] improved this method by organizing the PM as a vertex hierarchy instead of a linear sequence. The vertex hierarchy is known to provide very smooth LOD transitions, while requiring high runtime computations. This technique has been extended to large-scale polygonal models that consist of hundreds of millions of triangles [YSGM04]. However, VDR has not been applied widely to articulated models for large-scale crowd rendering.

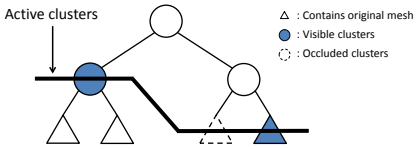


Figure 3: This figure shows our cluster hierarchy of our VDR-AM representation with active clusters (i.e. LOD cut) including visible and occluded clusters. Note that leaf clusters contain the original mesh.

Visibility culling also has been well studied to reduce the number of rendered triangles in scenes that have a high depth complexity [COCSD03]. For general environments, image-based occlusion representations are widely used, and high-performance culling algorithms use GPUs to perform occlusion culling [YSM03].

Hybrid algorithms for rendering acceleration. Many hybrid algorithms [RL00, YSGM04, CBWR07] that combine simplification with visibility culling for static polygonal models have been proposed. These techniques have integrated various visibility techniques into VDR frameworks of static polygonal meshes. Our VDR-AM representation can enable hybrid rendering methods for articulated models.

3 OVERVIEW OF OUR METHOD

In this section we explain our representation, followed by an overview of its construction and our runtime rendering algorithm.

3.1 Dual Representation

We use a *cluster hierarchy* (Fig. 3) for an articulated model, as a dual representation. The cluster hierarchy serves both as a multi-resolution hierarchy for View-Dependent Rendering (VDR), and as a bounding volume hierarchy for occlusion culling, view-frustum culling, and other geometric operations.

We call each node of the hierarchy a *cluster*. Each cluster serves as a main processing unit for VDR and occlusion culling. Each leaf cluster of the hierarchy contains a portion of the base mesh of the articulated model. For an intermediate cluster of the hierarchy, we merge two sub-meshes contained in its two child clusters, simplify them, and store them in the intermediate cluster. Therefore, each leaf cluster can provide the original resolution of a portion of the mesh, while an intermediate cluster can provide a low resolution for a portion of the mesh. Also, each cluster is associated with a bounding volume that contains all the geometry throughout the animation.

Each cluster records its maximum geometric simplification error that is caused by simplifying the sub-mesh of the cluster, while considering poses of an animation for the articulated model. In order to allow drastic simplifications on the articulated model given an error bound,

it is critical to cluster vertices that have similar simplification errors, thus leading to smaller geometric simplification errors for each cluster. In order to provide a high culling ratio, each cluster should be constructed such that it contains a spatially coherent portion of the base mesh of the articulated model.

3.2 Construction

In order to construct the cluster hierarchy of an articulated model, we first decompose the base mesh of the model into clusters, which become the leaf clusters of the hierarchy. We perform our error-aware clustering method to construct each cluster to have a spatially-coherent portion of the mesh whose vertices have similar simplification errors (Sec. 4.2). For the simplification of sub-meshes contained in clusters, we use the well-known edge-collapse and quadric-based simplification methods, which also consider poses of the animation of the model. Since the simplification process can take a large amount of time for the animation that consists of many poses, we propose a pose selection method that chooses representative poses for the animation and simplify the mesh by considering only those representative poses (Sec. 4.1).

3.3 Rendering Algorithm

To show benefits of our VDR-AM representation, we apply it to view-dependent rendering integrated with occlusion culling for articulated models. At runtime, we compute a new position and orientation (e.g., animation pose) of each articulated model in the scene. To perform VDR, we maintain *active clusters* (i.e. a LOD cut) that represent the articulated model with the lowest number of triangles, given a user-specified error bound (Sec. 5.1). To determine active clusters, we compute a screen-space projected simplification error for each cluster and compare it with the user-specified error bound. To perform occlusion culling we compute a conservative set of visible clusters based on the bounding volume information encoded in the cluster hierarchy and render them for the final images (Sec. 5.2).

4 CLUSTER HIERARCHY CONSTRUCTION

In this section we explain our cluster hierarchy construction method. We also compute clusters from the static polygonal models of scenes, as we compute clusters from the animated, articulated models.

Pose space reduction. Since an articulated model can have many bones (e.g., 30 to 60 bones for human-like characters), each pose can be considered as a point in a high dimensional pose space. Unfortunately, any operations on these high dimensional points can be very difficult and expensive because of the well-known curse of dimensionality. To ameliorate this issue, we reduce the dimensionality of the pose space. In particular, we use the multi-dimensional scaling, a statistical tool for reducing dimensions of data [McG68]. This method maps data into a reduced dimensional space, while preserving the distance between data in the original dimensional space. For all the tested models in the paper, we reduce the pose space of such models to 9 dimensional space, as suggested by Assa et al. [ACCO05]. Though Assa et al. suggested mainly for human-like models, we found that 9 dimensional space works well to other tested animal models.

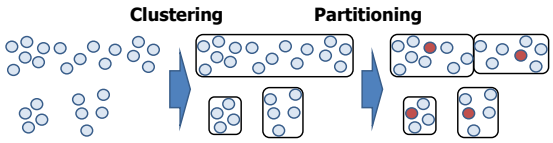


Figure 4: This figure shows the process of our pose selection method that performs the clustering and partitioning. The red points are the selected final poses that represent the distribution of the original poses.

4.1 Representative Pose Selection

In order to reduce the time taken on simplifying models, we propose to consider only *representative poses* instead of considering all the original poses during the simplification process. As the number of representation poses decreases, we can improve the performance of the simplification, but can underestimate the simplification error more, leading to a low-quality simplification.

To minimize this negative effect, we define *pose groups*, each of which contains a coherent set of poses. We then choose a representative pose from each pose group. After reducing the dimension of the pose space, we partition poses into pose groups based on our *clustering and partitioning* framework (Fig. 4).

We start with creating a pose group for each pose point. Our pose selection method consists of two stages: clustering and partitioning stages. In the clustering stage, we recursively merge two groups into a new pose group, if any two points chosen from those two pose groups are within the Euclidean distance of d. In order to efficiently perform this operation, we can construct a kd-tree from poses in the reduced pose space and find nearest neighboring pose points [Ben75].

The pose groups computed from the clustering stage can be quite big (e.g., the top group computed from the clustering step shown in Fig. 4), since we compute pose groups based only on the local distance information between poses. We adopt a second, partitioning stage that splits big groups into smaller groups. More specifically, we check whether the diagonal size of the bounding box computed from a pose group, c, is bigger than a threshold (e.g., $1.5 \times d$). If so, we recursively split the group c into two pose groups by using a median spatial partitioning, which divides the longest width of the bounding box of the group c into half. For each final pose

Figure 5: This figure shows eight poses chosen out of 35 poses for the walking animation based on our pose selection method.

group, we choose a representative pose (e.g., red circles shown in Fig. 4) that is closest to the center of the group.

We tried a well-known clustering method, K-means, for computing representative poses. We chose our clustering and partitioning framework instead of K-means, mainly because it is hard to set the target number of pose groups that well represents the distribution of the original poses for different animation patterns (e.g., walking, running, etc.).

4.2 Error-Aware Clustering Method

To construct the cluster hierarchy of an articulated model, we first decompose the base mesh of the model into a set of clusters. These computed clusters become leaf clusters of the cluster hierarchy.

We identify two different criteria and consider them for the cluster construction. First, each cluster should be a spatially coherent portion of the mesh, in order to keep the bounding box of the cluster small and thus achieve a high culling ratio. Second, each cluster should contain vertices that have similar simplification errors. For example, if a cluster contains two different mesh regions such as a joint and upper arm, highly deforming and less deforming regions respectively in the animation, then the cluster can get a high simplification error caused by the joint region, even though some portions (e.g., the upper arm) contained in the cluster may have a much lower simplification error. In this case, we may have to render a higher number of triangles even though some of those triangles can be simplified further given the user-specified error bound.

We, therefore, propose an error-aware clustering method that considers the simplification error as well as the spatial coherence for the sub-mesh contained in each cluster. To consider the simplification error for the geometry contained in each cluster during clustering, we have to simplify the mesh and compute simplification errors. However, this causes a chicken-and-egg problem, since we have to construct clusters first before simplifying clusters.

In order to avoid this problem, we measure how much a vertex deforms during the animation as a *deformation level* for the vertex, and use it as a rough approximation of the simplification error for the vertex. This is because as a vertex deforms more, it tends to generate a higher simplification error computed by considering different poses of animations.

Figure 6: This figure shows two images of an office evacuation crowd scene with two hundred articulated characters. This scene consists of 16.4 M triangles. Our method can achieve 49 frames per second (fps) on average for this scene.

To measure the deformation level of a vertex, we identify a bone, b_a, that is a least common ancestor for bones that affect the vertex and then compute a tightest bounding box that contains the trajectory of the vertex during the animation in the reference frame of the bone b_a. The deformation level of the vertex, then, is computed as the diagonal length of the bounding box. The computed deformation level does not fully capture the simplification error, which is also affected by the neighboring triangles of the vertex. Nonetheless, we have found that it works well for our tested benchmarks and is very easy to compute the deformation level during the clustering stage.

For computing clusters of the base mesh, we adopt a clustering and partitioning framework that is similar to the one we used for computing pose groups. In the first clustering stage, we group adjacent vertices (i.e. spatially coherent vertices) with similar deformation levels into a cluster. We define that two vertices have the similar deformation levels, when their deformation levels differ within the range of 10%. We continue this process until we cannot merge vertices any more.

Clusters computed from the clustering stage can have a very high number of triangles. Therefore, we apply the partitioning stage, which recursively splits clusters that have more than s (e.g., 100) triangles based on the median-based spatial partitioning. Fig. 7-(b) shows computed leaf clusters based on our method for an input model.

4.3 Hierarchy Construction

We construct the cluster hierarchy in a bottom-up manner, starting from leaf clusters computed in Sec. 4.2. We construct the hierarchy by merging two adjacent clusters that have similar deformation levels. We define two clusters to be adjacent, if they have adjacent triangles that share the same vertices. We also define the deformation level of a cluster to be the maximum of the deformation levels of vertices contained in the cluster.

Particularly, among adjacent clusters for a cluster, c, we identify a cluster, c_m, that has the minimum difference

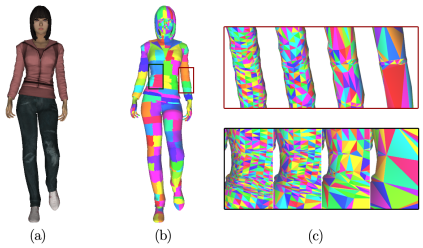


Figure 7: This figure shows an original model (a) and its leaf clusters generated by our error-aware clustering method (b). (c) show zoomed regions of boxed parts in (b). The top of (c) shows a deforming elbow region with increasing simplification levels. The bottom shows a non-deforming torso region, resulting in more aggressive simplification.

of the deformation level to that of the cluster c, and merge c_m and c into their parent cluster. We repeat this procedure until we construct the root node of the cluster hierarchy.

Simplification. Once we construct the hierarchy, we then perform the simplification for each cluster by traversing clusters in a bottom-up manner. For each leaf cluster, we simplify the sub-mesh contained in the cluster such that the number of the triangles in the cluster is reduced into half. For the simplification, we apply the pose-independent simplification method [DR05] that uses the well-known quadric simplification error metric. During the simplification, we consider only representative poses computed by our pose selection method (Sec. 4.1).

5 GPU-BASED RENDERING

In this section we explain how we can design an interactive GPU-based VDR method integrated with occlusion culling based on our VDR-AM representations for articulated models.

5.1 LOD Selection

In order to perform VDR and culling, we first compute a LOD cut in the cluster hierarchy that represents the articulated model given the error bound. To compute the LOD cut, we measure the geometric simplification error of each cluster in the screen space. To do that, we pre-compute a sphere whose diameter corresponds to the maximum Hausdorff distance between the original mesh and its corresponding simplified mesh contained in each cluster. For efficient computation, the maximum Hausdorff distance is approximated as the square root of the maximum quadric error associated with each cluster. We project the sphere associated with each cluster to the screen space. If the diameter of the projected sphere is equal to or smaller than the user specified pixels-of-error (PoE) value, we treat the cluster to have an enough resolution that can represent the articulated model given the PoE value. This LOD cut selection method takes only a minor CPU time (e.g., less than 1 ms) even for the exhibition crowd scene (Fig. 1) that consists of 1 K articulated models and 83 M triangles.

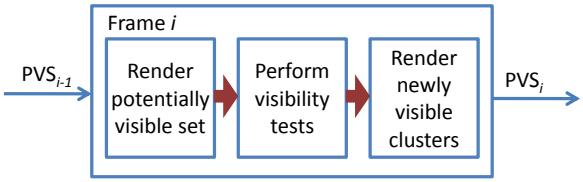


Figure 8: This figure shows our runtime rendering architecture.

5.2 Rendering Algorithm

We use an image-based conservative culling algorithm [YSM03] that is based on the frame-to-frame coherence and hardware-accelerated occlusion queries, in order to achieve an efficient performance of culling. We briefly explain how we enable such image-based culling based on our VDR-AM representation for articulated models.

Before we perform the culling, we decompose active clusters of the hierarchy into two disjoint sets: *potentially visible set* (PVS) and *potential occludee set*. Clusters in the PVS are treated to be visible and are used to create an occlusion map for visibility tests of clusters. We use PVS_i to denote the PVS at frame i. The overall architecture of our rendering algorithm is shown in Fig. 8.

Occlusion map generation. At frame i, we start with PVS_{i-1}, the PVS at the previous frame $i-1$. As Step 1 of our algorithm, we first update, i.e. simplify or refine, clusters of PVS_{i-1} to meet the error bound with our LOD selection method (Sec. 5.1) that considers the current view information. We set those clusters as PVS_i. We then render clusters of PVS_i into color and depth buffers. The depth buffer information computed with PVS_i serves as an occlusion map for visibility tests in the next rendering step.

Visibility tests. All the clusters of the LOD cut that are not in the PVS_i are used for the potential occludee set. We also update all the clusters in the occludee set. In Step 2 of our rendering algorithm, we check the visibility of clusters of the occludee set by using hardware accelerated occlusion queries. For clusters that passed the standard view-frustum culling, we use the bounding volumes of those clusters with the occlusion queries against the occlusion map. These bounding volumes serve as a conservative proxy to the geometry contained in corresponding clusters. We call those clusters of the occludee set that are identified as visible clusters with occlusion queries *newly visible* clusters. Fig. 9 shows an example of culling results in one of our benchmark scenes.

Rendering newly visible clusters. As the final step of our algorithm, we render newly visible clusters, to

Figure 9: The left figure shows the image created at a first person view. The right image shows a third person view with occlusion results. The black lines show the view frustum of the first person view. The pink, blue, and yellow boxes are visible, occlusion culled, and view-frustum culled clusters.

create the final image. Also, these newly visible clusters are added to the PVS_i. We use the PVS_i for the next frame. Note that by taking advantage of temporal coherence, we only update and reset the PVS every n frames.

Instancing and static models. It is common to use instancing to create large-scale crowd scenes. We also store various data of cluster hierarchies such that we can efficiently utilize the GPU-based instancing. More specifically, we adopt the pseudo-instancing method [Zel04] for our VDR representation. In addition, our VDR-AM representation can be easily applied to handling static models. In this case our method considers only base meshes of static models. As a result, our rendering method can be applied to handling both static and articulated models.

6 RESULTS

To show benefits of our representation, we have implemented our construction and runtime rendering algorithms on a 3.00 GHz Intel quad-core PC with a GeForce 8800 GTX GPU that has 768 MB. We store all the transformation matrices of bones of various articulated models with all the poses in a 1 D texture buffer, which is easily accessible in the GLSL vertex shader that implements our runtime skinning method. For all the performance tests, we use the HD image resolution of 1280 by 720. In this image resolution, we use the PoE value of 0.5, in order to avoid visual artifacts to viewers. Since the used PoE causes only sub-pixel errors in the screen space, we do not perform any expensive geomorphing [Hop97].

Benchmark scenes. We have tested our method with three different crowd scenes that consist of human and animal articulated characters. Our first benchmark represents an office evacuation scenario (Fig. 6). This office scene consists of 200 instanced virtual human characters and 16.4 M triangles. Our second benchmark represents an exhibition scenario (Fig. 1). This exhibition scene consists of 1 K instanced characters and 83 M triangles. The third scene is a stampede scenario (Fig. 2) consisting of 5 K instanced animal characters and 242 M triangles. In the first and second crowd scenes, we use 20 different virtual human characters that have 35 bones and 73 K to 110 K triangles for their base meshes. Also, they are animated by using a walking animation pattern that consists of 35 different poses. In the third crowd scene, we use horse, elephant and camel models that consist of 17 K, 85 K, and 44 K triangles respectively. They have 30 to 40 bones and are animated in a running pattern with 15 to 80 poses.

Pose selection parameter setting. In our pose selection method, the parameter d trades-off between the simplification quality and performance of our overall construction method. If we set d to be too high, we would choose too small number of representative poses, causing faster construction, but leading to a worse simplification quality. On the other hand, if we set d to be too low, we would get the reverse effects: slower construction, but higher simplification quality. Given this trade-off space, we found that the range of 10% to 30% of the diagonal size of the bounding box of the model for d strikes a good balance in our tested benchmarks.

In this setting, for a walking animation that consists of 35 poses, our method selects 8 different poses. For a snake crawling animation with 81 poses, our method selects 13 poses. As a result, our method achieves 3 to 5 times performance improvement for our construction method. Also, in terms of the simplification quality, we found that the RMS distance between the original mesh and the simplified mesh computed only from considering computed representative poses is within 1% to 2% difference to the RMS distance between the original mesh and the simplified mesh computed from considering all the poses; we use the metro tool [CRS98] to measure the RMS distance. Fig. 5 shows eight chosen poses out of 35 poses of the walking animation. Note that more poses are chosen during a period that the human character switches his pivoting leg.

Hierarchy construction comparisons. In order to show the benefits of our error-aware clustering method, we additionally implemented a naive clustering method that uses an octree. More specifically, we construct clusters by recursively partitioning the base mesh until the sub-mesh contained in each node of the octree has s triangles. This naive method considers only the base mesh computed from a pose and does not consider any simplification errors. Once we compute clusters from the octree, the hierarchy construction and simplification that we have applied to to our error-aware clustering method are performed in the same manner to the naive, octree-based clustering method.

Compared to this naive method, our error-aware clustering method shows only 4% slower clustering performance at preprocessing, but shows 50% higher runtime rendering performances, by rendering 50% fewer triangles given the same error bound in our tested benchmarks. Since we cluster vertices that have similar sim-

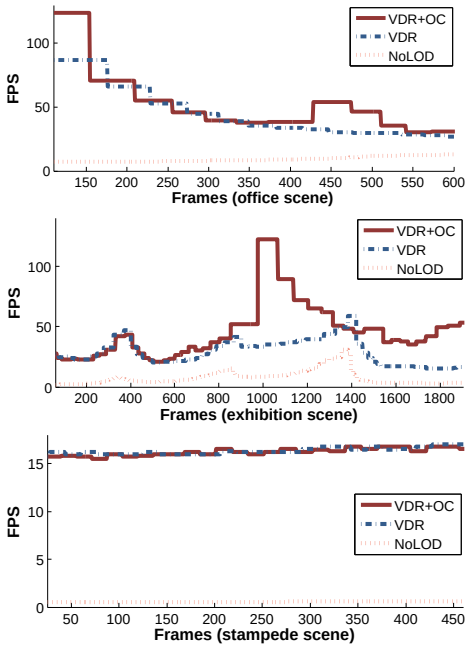


Figure 10: These figures show fps graphs of different methods: our VDR method, **VDR**, our VDR method integrated with occlusion culling, **VDR+OC**, and **NoLOD** that uses the original resolutions and view-frustum culling.

plification errors, we can allow more drastic simplification if possible, over the naive octree-based clustering method. Fig. 7-(c) shows our simplification results on different portions of a walking character.

Construction time and memory requirement. We set each cluster to have less than 100 triangles. In this setting, our method creates 1.8 K leaf clusters and takes 19 min in CPU to compute our representation for the biggest virtual character that has 110 K triangles. Also, our representation requires 72 bytes per each triangle for an articulated model. Since the original mesh requires 30 bytes per each triangle, our representation requires 140% more space over the original mesh. Since our representation stores simplified triangles, whose number is similar to the number of original unsimplified triangles, our representation requires at least 100% more space over the original mesh. Therefore, the memory overhead 140% of our representation is not significantly high.

Comparison configurations. We measure the performance of three different methods: 1) a base rendering system, **NoLOD**, that uses the original resolutions and performs only view-frustum culling, 2) our VDR rendering system, **VDR**, that performs the VDR on the base system, and 3) our VDR system integrated with occlusion culling, **VDR+OC**. We measure the frames per second (fps) of these methods with pre-defined paths, which are shown in the accompanying video. The fps graphs of these methods with our benchmark scenes are shown in Fig. 10.

Benchmark	FPS			#. Rendered Tri. (M)		
	NoLOD	VDR	VDR+OC	NoLOD	VDR	VDR+OC
Office Fig. 6	10.27	43.29	49.21	12.46	2.29	2.06
Exhibition Fig. 1	7.73	29.15	46.08	24.66	3.14	1.94
Stampede Fig. 2	0.55	16.38	16.23	148.09	4.77	4.70

Table 1: This table shows the frames per second (fps) and the number of rendered triangles on average while rendering scenes. **NoLOD**, **VDR**, and **OC** represent rendering with the original resolutions, our view-dependent rendering method, and our occlusion culling technique respectively.

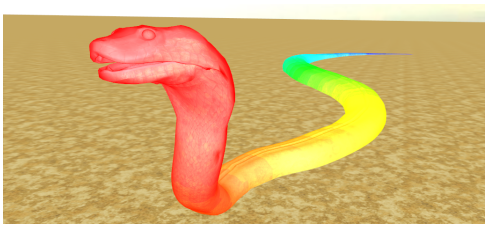


Figure 11: This figure shows adaptively varying resolutions computed from our VDR-AM representation. We render each vertex of the model with colors computed from the well-known heat color map; the red color indicates the highest resolution, while the blue color refers to the lowest resolution.

6.1 Rendering Performance

On average, **VDR+OC** achieves interactive performances, 49 fps and 46 fps in the office and exhibition crowd scenes respectively (Table 1). In the exhibition scene, compared to the base rendering system, we achieve 4 times performance improvement by enabling the VDR, and achieve 6 times performance improvement by enabling VDR and occlusion culling together. We also observe a similar performance gain with the office scene. These performance improvements are mainly caused by reducing the number of triangles that we have to process for the skinning and rendering operations. More specifically, the base rendering system renders 458 K clusters that contain 24.66 M triangles for the exhibition scene. On the other hand, **VDR** renders 12 K clusters with 3.14 M triangles and **VDR+OC** renders 8 K clusters with 1.94 M triangles.

In the stampede scene our method achieves interactive performance, 16 fps, even though the original model consists of more than 200 M triangles. Also, it demonstrates high performance improvement (up to 30 times) by enabling VDR over the base rendering method. Nonetheless we show a minor, but lower performance by enabling occlusion culling over VDR. This is mainly because we do not have much depth complexity in the tested view point.

6.2 Discussions

To highlight the benefit of our method, we show varying resolutions of our view-dependent representation for a

snake articulated model that consists of 28 K triangles and 81 poses for its crawling animation (Fig. 11). We use the heat color map to show how the resolution of the model varies given the first person view. As a portion of the model is farther away from the viewer, it gets lower resolution as indicated by showing colors close to the blue one. At the given view our method requires only 18 K triangles for rendering the snake model.

Relationships with LODs. Many view-dependent representations have been proposed, as discussed in Sec. 2. These techniques have not been widely applied to articulated models. This may be mainly because that many prior view-dependent techniques have high computational overheads. Nonetheless, our technique reduces the computational overhead by providing view-dependent resolution at a granularity of clusters consisting of around 100 triangles, not each triangle of the mesh. Note that this kind of approach is inspired by efficient LOD rendering techniques such as HLODs (Hierarchical levels of detail) [EMB01] designed for large-scale static models.

Breakdown of each rendering component. We also measure how much percentage each component of our method takes over the total rendering time. The CPU-based selection of the LOD cut given a viewing configuration takes less than 1.5 ms. The GPU-based skinning, rendering, and occlusion culling components take 14%, 82%, and 4% over the total rendering time on average across all the tested benchmarks.

Limitations. Even though our method shows performance improvements over the base rendering method, there is no guarantee that our VDR integrated with occlusion culling always improves the performance of various crowd scenes. This is mainly because performing VDR and occlusion culling has overheads. Also, our method may show visual artifacts with PoE values bigger than 1, while we were able to achieve interactive performance without noticeable artifacts by using 0.5 PoE for our tested models. Geomorphing can ameliorate *popping* artifacts by providing smooth transitions between different LODs.

7 CONCLUSION

We have proposed view-dependent representation for articulated models, VDR-AM, and presented how we can use it for an interactive view-dependent rendering method integrated with occlusion culling in large-scale crowd scenes. VDR-AM consists of a cluster hierarchy that serves both as a multi-resolution representation for VDR and a bounding volume hierarchy for occlusion culling. We also presented an error-aware cluster construction method to allow drastic simplifications on portions of meshes of articulated models. We were able to achieve 16 to 49 fps on average for large-scale crowd scenes that consist of thousands of articulated models and hundreds of millions of triangles without noticeable visual artifacts.

There are many avenues for future work. In addition to addressing current limitations of our method, we would like to first handle larger crowd scenes by designing a more drastic simplification method (e.g., volumetric simplification methods) as well as to simplify skeletal and behavioral models of characters [RCBS10]. One can combine our method with image-based representations [DHOO05, KDC$^+$08]; use our method in a near field and use impostors allowing more drastic simplifications in a far field. Also, we used deformation levels to estimate simplification errors. We would like to extend this concept to identify vertices that have similar rotational sequences [JT05], to more accurately cluster vertices that have similar simplification errors. In addition, it would be interesting to conduct a user study measuring perceptual errors of our method. Finally, we would like to apply our VDR-AM to other geometric applications such as collision detection. Since our representation is based on a polygonal representation, we believe that it can be easily applied to collision detection in a similar spirit to the collision detection method designed for dynamic simplification [YSLM04].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive feedbacks. This work was supported in part by MCST/KOCCA/CT/R&D 2011, MKE/KEIT [KI001810035261], MKE/MCST/IITA [2008-F-033-02], BK, DAPA/ADD (UD110006MD), MEST/NRF/WCU (R31-2010-000-30007-0), KMCC, MSRA, and MEST/NRF (2012-0009228).

8 REFERENCES

[ACCO05] Assa, J., Caspi, Y., and Cohen-Or, D., Action synopsis: pose selection and illustration. ACM Trans. Graph., 24, no. 3, pp. 667–676, 2005.

[Ben75] Bentley, J.L., Multidimensional binary search trees used for associative searching. Communications of the ACM, 19, pp. 509–517, 1975.

[CBWR07] Charalambos, J.P., Bittner, J., Wimmer, M., and Romero, E., Optimized hlod refinement driven by hardware occlusion queries. In Advances in Visual Computing, Springer, 2007, pp. 106–117.

[COCSD03] Cohen-Or, D., Chrysanthou, Y., Silva, C., and Durand, F., A survey of visibility for walkthrough applications. IEEE Tran. on Visualization and Computer Graphics, 9, pp. 412–431, 2003.

[CRS98] Cignoni, P., Rocchini, C., and Scopigno, R., Metro: Measuring error on simplified surface. Computer Graphics Forum, pp. 167–174, 1998.

[DHOO05] Dobbyn, S., Hamill, J., O'Conor, K., and O'Sullivan, C., Geopostors: a real-time geometry/impostor crowd rendering system". ACM Transactions on Graphics, 24, no. 3, pp. 933–933, 2005.

[DR05] DeCoro, C., and Rusinkiewicz, S., Pose-independent simplification of articulated meshes. In Symp. on Interactive 3D Graphics, 2005, pp. 17 – 24.

[EMB01] Erikson, C., Manocha, D., and Baxter, B., Hlods for fast display of large static and dynamic environments. Proc. of ACM Symposium on Interactive 3D Graphics, 2001.

[GH97] Garland, M., and Heckbert, P., Surface simplification using quadric error bounds. ACM SIGGRAPH, pp. 209–216, 1997.

[HCC06] Huang, F.C., Chen, B.Y., and Chuang, Y.Y., Progressive deforming meshes based on deformation oriented decimation and dynamic connectivity updating. In ACM Symp. on Computer Animation, 2006, pp. 53–62.

[Hop97] Hoppe, H., View dependent refinement of progressive meshes. In ACM SIGGRAPH, 1997, pp. 189–198.

[JT05] James, D.L., and Twigg, C.D., Skinning mesh animations. ACM Trans. on Graphics (SIGGRAPH), 24, no. 3, 2005.

[KDC+08] Kavan, L., Dobbyn, S., Collins, S., Zara, J., and O'Sullivan, C., Polyposters: 2d polygonal impostors for 3d crowds. In ACM Symp. on Interactive 3D Graphics and Games, 2008, pp. 149–155.

[KG05] Kircher, S., and Garland, M., Progressive multiresolution meshes for deforming surfaces. In Symp. on Computer animation, 2005, pp. 191–200.

[LRC+02] Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., and Huebner, R., Level of Detail for 3D Graphics. Morgan-Kaufmann, 2002.

[LS09] Landreneau, E., and Schaefer, S., Simplification of articulated meshes. Computer Graphics Forum, pp. 347–353, 2009.

[McG68] McGee, V.E., Multidimensionnal scaling of n sets of similarity measures : A nonmetric individual differences approach. Multivariate Behavioral Research, 3, pp. 233–248, 1968.

[MG03] Mohr, A., and Gleicher, M., Deformation sensitive decimation. University of Wisconsin Graphics Group. Technical Report, 2003.

[NGCL09] Narain, R., Golas, A., Curtis, S., and Lin, M.C., Aggregate dynamics for dense crowd simulation. In SIGGRAPH Asia, 2009, pp. 1–8.

[PSA07] Pilgrim, S., Steed, A., and Aguado, A., Progressive skinning for character animation. Computer Animation and Virtual Worlds, 18, no. 4-5, pp. 473–481, 2007.

[RCBS10] Rodriguez, R., Cerezo, E., Baldassarri, S., and Seron, F.J., New approaches to culling and lod methods for scenes with multiple virtual actors. Computers and Graphics, 34, no. 6, pp. 729 – 741, 2010.

[RD05] Ryder, G., and Day, A.M., Survey of real-time rendering techniques for crowds. Computer Graphics Forum, 24, no. 2, pp. 203–215, 2005.

[RL00] Rusinkiewicz, S., and Levoy, M., Qsplat: A multiresolution point rendering system for large meshes. SIGGRAPH, pp. 343–352, 2000.

[TCYM09] Tang, M., Curtis, S., Yoon, S.E., and Manocha, D., Iccd: Interactive continuous collision detection between deformable models using connectivity-based culling. Visualization and Computer Graphics, IEEE Transactions on, 15, no. 4, pp. 544 –557, 2009.

[TOY+07] Thalmann, D., O'Sullivan, C., Yersin, B., Maim, J., and McDonnell, R., Populating virtual environments with crowds. In Eurographics Tutorial, 2007.

[YGKM08] Yoon, S.E., Gobbetti, E., Kasik, D., and Manocha, D., Real-Time Massive Model Rendering. Morgan & Claypool Publisher, 2008.

[YSGM04] Yoon, S.E., Salomon, B., Gayle, R., and Manocha, D., Quick-VDR: Interactive View-dependent Rendering of Massive Models. In IEEE Visualization, 2004, pp. 131–138.

[YSLM04] Yoon, S., Salomon, B., Lin, M.C., and Manocha, D., Fast collision detection between massive models using dynamic simplification. In Eurographics Symposium on Geometry Processing, 2004, pp. 136–146.

[YSM03] Yoon, S., Salomon, B., and Manocha, D., Interactive view-dependent rendering with conservative occlusion culling in complex environments. In Proc. of IEEE Visualization, 2003.

[YYBE13] Yuksel, K., Yucebilgin, A., Balcisoy, S., and Ercil, A., Real-time feature-based image morphing for memory-efficient impostor rendering and animation on gpu. The Visual Computer, 29, pp. 131–140, 2013.

[Zel04] Zelsnack, J., Glsl pseudo-instancing. Tech. rep., NVIDIA Corporation, 2004.

Texture Mapping of Images with Arbitrary Contours

Nicolas Cherin Frederic Cordier Mahmoud Melkemi
LMIA, Université de Haute Alsace (LMIA, EA 3993)
4, rue des Frères Lumière
68093, Mulhouse, France
nicolas.cherin@uha.fr frederic.cordier@uha.fr mahmoud.melkemi@uha.fr

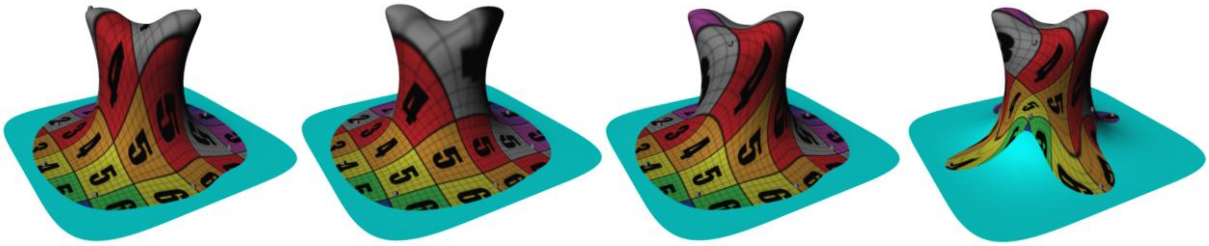


Figure 1. Local texture mapping with different sets of feature points.

ABSTRACT

Decaling is an intuitive paradigm for texture mapping in an analogy of attaching stickers on an object in the real world. This paradigm enables an artist to put decals directly on a 3D model after interactive manipulations such as modifying their positions, scales and orientations. In this paper, we present a novel method for multiple-constrained decaling. Given a region inside a texture together with a set of feature points in the region and a 3D model, our problem is to map the texture region onto the surface of the model in an intuitive manner, while satisfying the constrained imposed by a user-specified correspondence between a set of feature points in the region and the surface. We propose a solution for this problem. Our approach iteratively determines a portion of the mesh representing the surface while accordingly refining its parameterization, guided by the feature point correspondence.

Keywords

Texture Mapping, Parameterization, Polygonal Modeling.

1. INTRODUCTION

Texture mapping is a well-known technique for mapping an image onto the surface of a 3D model to enhance its visual appearance. This technique has been adopted for a broad range of applications such as special effects for the film industry that requires highly realistic models as well as the game industry for efficiently creating 3D models and virtual characters. The essential step of texture mapping is the surface parameterization of a 3D model, i.e. finding a one-to-one correspondence between the entire surface of the model and a texture.

A 3D model can also be decorating with several textures, that is, different images of arbitrary shapes are placed on the 3D model, each image covering a portion of the surface by locally parameterizing the region on the 3D surface that corresponds to each image. The metaphor can regarded as affixing stickers or decals to the surface of the model. This technique shows the possibility of texturing models by compositing images directly on the 3D surface, which is analogous to 2D image compositing that creates a new image by combining images from different sources by alpha blending. The texture mapping with local parameterization usually produces higher quality results than texture mapping with global parameterization, since local parameterization for a smaller number of triangles results in lower distortion compared to global parameterization for the entire surface.

The latest work related to local parameterization uses a discrete approximation to the exponential map [Sch06a] that parameterizes a circular region around a center point provided by the artist. As pointed by the authors, a disadvantage of their technique is that

the distortion of mapping increases significantly as the textured region is becoming larger, especially on surfaces with high frequency features. In addition, their technique offers limited control, that is, only the position of the center point, the scaling and the orientation of local parameterization can be specified by the artist. This method cannot be used for constrained texture mapping where we need to define a correspondence between multiple feature on the image and the 3D surface.

We present a method which is a generalization of the work of Schmidt et al., that is, the mapping of an image of an arbitrary shape onto the 3D surface given multiple corresponding pairs of feature points on the image and the surface. The input of our approach is a region of the 2D image that is bounded by a simple closed curve, and a set of feature points in the region and their counterparts on the 3D model. Our method computes automatically the region for texturing on the 3D model and its parameterization in an intuitive manner. Compared to the work of Schmidt et al., our approach offers two important advantages. First, the artist can use as many feature points as needed; our method ensures the exact matching of the features between the image and the 3D model guided by the feature point correspondence. Second, our method does not impose any limitations on the size and shape of the textured region on the model surface. Our method provides valid parameterization even when the textured region is very large and the surface of the model contains sharp features. For efficient texture mapping, we introduce a novel two-step parameterization that supports multiple feature correspondence and automatic computation of the 3D region for texturing. We show how to use this method to texture-map parts of the surface of a 3D model.

2. RELATED WORK

A variety of techniques have been proposed to help artists to decorate 3D models. We give a brief description of these techniques.

Surface Painting: Surface painting is one of the most common techniques for decorating 3D models; the artist creates a texture from scratch by drawing directly on a 3D model using painting tools such as a brush or an eraser. This technique has been well studied [Car04a] and many commercial tools for 3D painting are available [May05a]. However, surface painting is tedious and requires artistic skills to create a complete texture.

Texture Tiling: Another technique for creating textures on 3D model is to cover its surface with partially overlapping images [Pra00a] [Tur01a] [Wei01a] [Sol02a]. This technique is useful for completely covering a surface with repetitive applications of a pattern image. However, the use of this method is limited since it can only be applied for texture tiling.

Global Planar Parameterization: A large body of work on texture mapping has been devoted to global parameterization of surfaces, i.e. finding a bijective function between the entire surface of a model and a planar texture space. If the surface is topologically equivalent to a disk, then a planar parameterization is computed through an optimization that finds the position of vertices in the texture space such that distortion of the triangles is minimized [San01a], [Lev02a], [Des02a], [Flo03a], [Flo03b], [Mey02a].

If the surface of the model is not topologically equivalent to a disk, the surface is segmented into a set of disjoint charts, each of which is homeomorphic to a disc and parameterized independently of each other [Lev02a], [Zho04a], [Zha05a]. These parameterized charts are then packed into the texture space to collectively form a texture atlas.

The surface parameterization technique has been further extended to incorporate a feature correspondence between points in the texture space and vertices on the surface of the model. The feature correspondence is integrated in parameterization either as soft constraints [Lev01b] [Des02a] or hard constraints [Kra03a]. [Zho05b] further extends the constrained parameterization to allow the artist to generate a texture atlas from multiple images.

Since global parameterization is targeted for mapping the entire surface, distortion due to parameterization usually increases with greater surface complexity. Our approach is based on local parameterization, which aims at lowering the distortion by restrictively parameterizing the portion of the surface that is actually textured.

Local Parameterization: Unlike global parameterization, local parameterization is computed only for the region of the 3D surface that is to be textured. This technique is known as decal mapping, in reference to the metaphor of a decal (or sticker) affixed to the surface of an object. [Lev05a] have proposed a system supporting the interactive manipulation and composition of decals. The parameterization of decals is obtained with a planar projection of the 3D surface to be textured. While this method is simple and shows good computational efficiency, the planar projection of highly-curved surfaces results in significant distortion. [Sch06a] have computed the parameterization with discrete exponential maps, which significantly improves the quality of the parameterization compared to the planar projection. Still, the quality of the parameterization with this method is sensitive to high frequency features of the 3D surface to be

parameterized. Our system combines conformal mapping and 2D warping to robustly handle the parameterization of surfaces with high frequency features. Besides, our method allows users to introduce multiple feature constraints, which facilitates precise alignment between texture and surface features.

Recently, some researchers [Sun13a] have proposed an interactive interface for texturing 3D surfaces. With this system, the user specifies a local parameterization with a free-form curve drawn on the surface. Compared to their method, our approach offers higher level of user interaction. In their system, the texture image should have the shape of a strip and its mapping is achieved through the manipulation of a surface curve. In our system, the texture image can be of any shape and the mapping is controllable with an arbitrary set of feature points.

3. OVERVIEW

We provide a novel texturing technique that is powerful, yet easy-to-use for decorating 3D models with one or more textures. Basic operations for mapping a texture on a 3D model are cutting an image with a simple closed curve and pasting the result onto the surface of 3D model guided by a set of corresponding pairs of feature points, each constraining a vertex of the 3D model to a position in the image.

For the remainder of the paper, we refer by the texture space to the 2D space where the image and the simple closed curve are located. The object space is the 3D space containing the triangular mesh of the model (Figure 2(b)). An image region is a part of the image that is surrounded by the closed curve (Figure 2(a)) and a patch is a subset of triangles of the triangular mesh, each triangle corresponding to a triangle in the 3D model to be textured. The position of the patch vertices in the texture space is computed through the parameterization of the patch.

Our texturing method is comprised of three steps. We first find a set of connected triangles on the 3D surface that contains all the feature points and that region containing the triangles is homeomorphic to a disc. We place these triangles inside the closed curve in the image (Figure 2(c)). The patch is then grown iteratively by adding a number of triangles at a time. At each iteration, we reparameterize the modified patch to minimize the texture distortion while satisfying the feature point constraints. This process is repeated until the patch completely covers the image region bounded by the closed curve (Figure 2(d)). Finally, we transfer the image region onto the 3D surface exploiting the patch and the parameterization (Figure 2(e)).

These three steps are described in details in the Sections 4, 5 and 6 respectively. Several examples of models textured with our tool are shown in Section 7. We discuss about the limitations and the future work in Section 8.

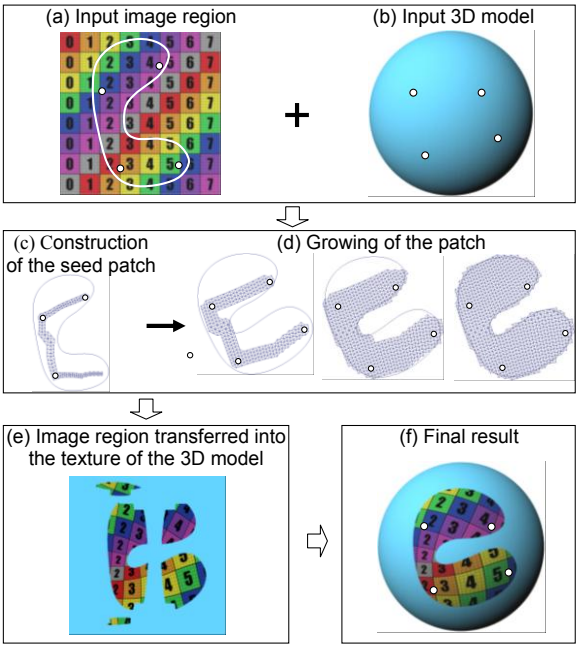


Figure 2. Overview of the texturing method.

4. BUILDING A SEED PATCH

The objective is to create a seed patch satisfying the feature constraints to bootstrap the patch growing. Specifically, the seed patch must be composed of a set of connected triangles containing the feature points; the position of feature points should be located at the given position and the vertices of each triangle in the patch should be inside the image region. In addition, since the image region is bounded by a simple closed curve, the patch should be homeomorphic to a disc.

In order to find the triangles on the 3D surface to build the patch, we first construct a 2D planar *feature-point graph* $\mathcal{G}_I$ in the image region; this graph has a set of vertices corresponding to feature points and a set of edges that are straight-lines joining a pair of feature points (see Figure 3(b)). Note that edges intersecting the boundary of the image region are not included in $\mathcal{G}_I$ (Figure 3(c)); the outline of $\mathcal{G}_I$ provides a rough approximation of the shape of the image region. We construct another graph $\mathcal{G}_S$ with the same connectivity as $\mathcal{G}_I$, on the 3D surface to obtain a rough approximation of the location of the patch on the 3D surface. Each edge of the 3D graph $\mathcal{G}_S$ corresponds to an edge of the 2D graph $\mathcal{G}_I$. Unlike a 2D edge, 3D edge represents the shortest Euclidean path in the triangular mesh that connects a pair of 3D feature points. Finally we use such path to find the triangles to construct the 3D patch.

4.1. Construction of the planar feature points graph in the image region

We first compute a triangulation of the feature points in the image region, employing a method which is essentially the same as the incremental Delaunay triangulation except that each edge of the triangulation lies completely in the image region. In order to identify the edges, we initially find all line segments, each connecting a pair of feature points and which do not intersect the boundary of the image region. These line segments are put in a priority queue ordered by their length. The line segments are then chosen one by one in sequence, starting from the shortest one, such that the line segments already chosen do not intersect each other. After adding an edge, we use edge-flipping algorithm [Hur99a], to flip edges which violate the local Delaunay criterion and do not intersect the boundary of the image region after the flip.

The resulting triangulation may be composed of several disconnected components due to the lack of line segments satisfying the non-intersecting requirement with the boundary. In this case, the artist is required to place additional feature points to obtain the triangulation.

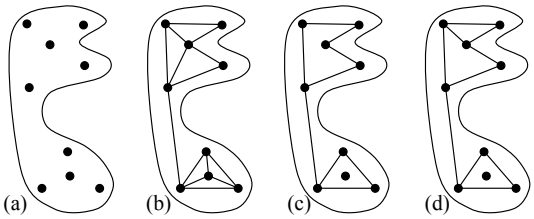


Figure 3. Construction of the feature point graph G_I: image region with the feature points (a), a triangulation of the feature points (b), removal of internal edges (c), decomposition into convex faces (d).

Next, we build a feature-point graph $\mathcal{G}_I$ by deleting all internal edges from the above triangulation (Figure 3(c)). Note that this may create feature points with no edges incident to it; these isolated feature points are always located inside a face. In order to simplify the computation of the initial parameterization of the seed patch (see section 4.3), we decompose every concave faces of $\mathcal{G}_I$ (regions bounded by edges) into convex ones by inserting additional edges (Figure 3(d)).

4.2. Selecting triangles to construct the seed patch

In this step, we identify the part of the surface of the 3D model to be textured. We first construct the graph $\mathcal{G}_S$ by embedding the edges of $\mathcal{G}_I$ onto the 3D surface. Given the 2D feature points $F_{I,j}$ and $F_{I,k}$ and their corresponding 3D feature points $F_{S,j}$ and $F_{S,k}$, the embedding of the edge $(F_{I,j},F_{I,k})$ is obtained by finding the shortest path on the mesh connecting $F_{S,j}$ to $F_{S,k}$.

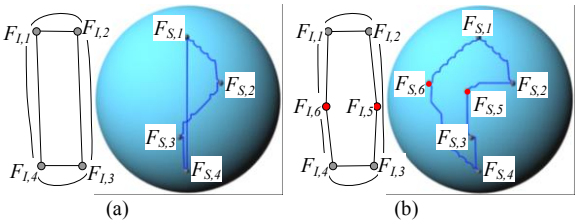


Figure 4. The paths of the face ($F_{S,1}$,$F_{S,2}$,$F_{S,3}$,$F_{S,4}$) intersect each other (a); two feature points $F_{S,5}$ and $F_{S,6}$ are placed such that the paths of the face ($F_{S,1}$,$F_{S,2}$,$F_{S,5}$,$F_{S,3}$,$F_{S,4}$,$F_{S,6}$) do not intersect each other (b).

Next, we test whether the paths belonging to the same face of $\mathcal{G}_S$ intersect each other (Figure 4(a)). If so, the artist inserts one or more feature points to avoid intersections between the paths as shown in Figure 4(b).

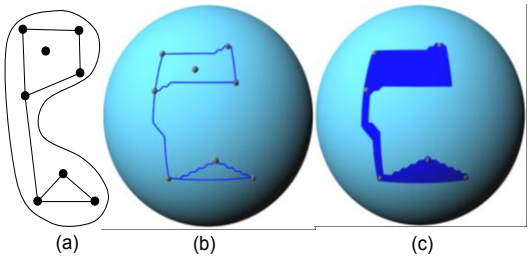


Figure 5. Construction of the seed patch: the graph G_I in the image region (a), the graph G_S on 3D surface (b), selection of triangles using G_S (c).

After constructing $\mathcal{G}_S$, we partition the mesh into regions along the paths of $\mathcal{G}_S$, each region corresponding to a face of $\mathcal{G}_S$ (Figure 5(b)). Next, the seed patch is constructed by merging (1) triangles of inner regions and (2) triangles encountered along the bridging paths (Figure 5(c)).

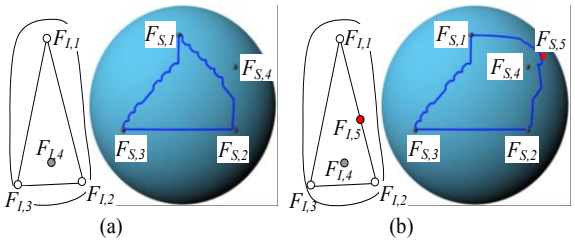


Figure 6. The feature point $F_{S,4}$ is outside the region bounded by ($F_{S,1}$,$F_{S,2}$,$F_{S,3}$) (a); $F_{S,5}$ is added such that $F_{S,4}$ is inside ($F_{S,1}$, $F_{S,2}$,$F_{S,3}$,$F_{S,5}$) (b).

Finally, we need to test if the acquired seed patch contains all the feature points. As previously stated, the graph may contain isolated feature points. Consider a 2D feature point $F_{I,4}$ and its corresponding 3D feature point $F_{S,4}$ as shown in Figure 6. Since $F_{I,4}$ is located inside a face, $F_{S,4}$

should be located inside the corresponding mesh region. If not, the artist is required to insert additional feature points as shown in Figure 6(b) so that both $F_{I,4}$ and $F_{S,4}$ are inside the face and the mesh region, respectively.

It is worth noting that a set of edges that do not form any face may self-intersect when embedded onto the 3D surface. This is useful for creating a self-overlapping texture on the mesh of the 3D model as shown in Figure 7. Mesh vertices located where the texture self-overlaps have several texture coordinates, each of which corresponding to a vertex in the patch.

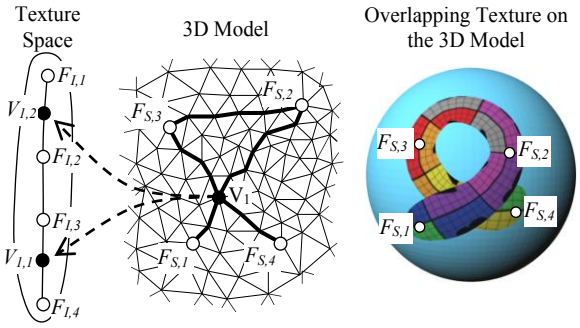


Figure 7. Self-intersecting paths on the 3D model: the vertex V_1 located at the crossing of the paths ($F_{S,1}$,$F_{S,2}$) and ($F_{S,3}$,$F_{S,4}$) on the 3D model has two corresponding vertices $V_{1,1}$ and $V_{1,2}$ in the patch.

4.3. Initial parameterization of the seed patch

Now that the seed patch has been created, we compute the texture coordinates of every vertex (Figure 8). Texture coordinates are computed with different methods, depending on whether the vertex is a feature point, belonging to a path connecting a pair of feature points or inside a face of the feature-point graph.

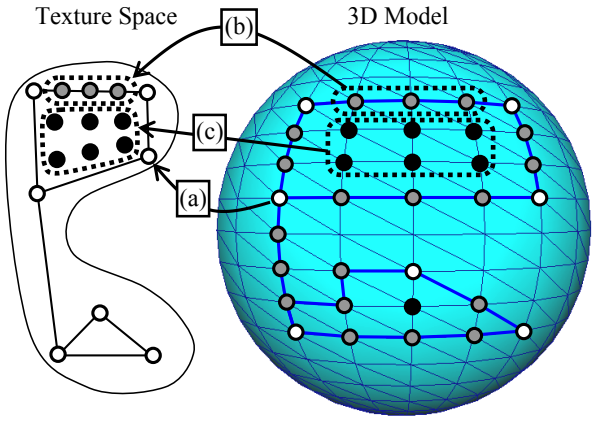


Figure 8. Initial parameterization of the seed patch.

Vertices corresponding to feature points are placed at the user-specified positions in the texture space (shown by the arrow (*a)* in Figure 8). Next, the vertices belonging to the paths of $\mathcal{G}_S$ are uniformly distributed along the corresponding edges of $\mathcal{G}_I$ (arrow (*b)* in Figure 8). Finally, other vertices belonging to a face of $\mathcal{G}_S$ are placed in the interior of the corresponding face of $\mathcal{G}_I$ (arrow (*c)* in Figure 8). Since the faces of $\mathcal{G}_I$ are convex, a simple parameterization method [Flo03b] can be used, which assumes the boundary of the parameterization to be fixed and convex. The initial parameterization of the seed patch may exhibit a high degree of distortion, but a precise parameterization is not critical to the final map, since the mapping will be modified in the subsequent steps.

5. GROWING THE PATCH

Our goal now is to find the complete set of triangles to be texture-mapped, along with their parameterization. These are achieved by growing the seed patch with its neighboring triangles and computing its parameterization in the texture space (see Figure 9).

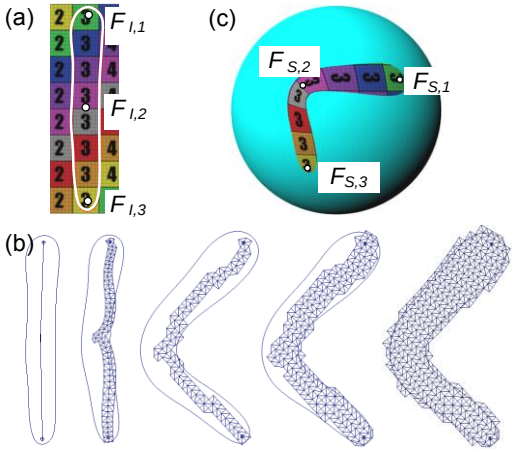


Figure 9. Warping of the image region: image region with the feature points (a); growing of the patch (b), the final model (c).

Note that the structure of the patch and its parameterization are inter-dependent: modifying the structure of the patch requires re-computing its parameterization. Likewise, if the parameterization changes, one needs to modify the structure. For instance, if any of the triangles lie outside the image region after re-computing the parameterization, they must be removed from the patch. Our proposed method adopts an incremental approach: At each iteration, the current patch is recomputed for its parameterization (Section 5.1), followed by an update of its structure (Section 5.2). This is repeated until the patch completely covers the given image region.

5.1. Parameterization of the patch

An unconstrained planar embedding of the patch is first computed using a conventional method. The boundary of the texture region may have any shape; therefore, we use the free-boundary conformal parameterization proposed by [Lev02a], because of its low computation time. Since the patch is grown incrementally, we compute the conformal parameterization using an iterative solver. Such an approach enables us compute the final parameterization with successive approximations starting from an initial parameterization of the patch; in addition, the iterations can be stopped whenever necessary. These features are required for the algorithm to grow the patch (see subsection 5.2.1)

Next we align the feature points of the patch with those of the image region, which can be achieved by warping either the planar embedding of the patch or the image region. In our work, we have chosen to warp the image region whose boundary polygon usually contains many less vertices than the patch (Figure 9).

We compute the warping using a method similar to the one proposed by [Seo10a] We construct a continuous 2D time-dependent vector field **v**(x,y,t) and obtain the new positions of a vertex *p* of the image region by applying a pathline integration of **v**(x,y,t) starting from *p*. Intuitively speaking, the vector field **v**(x,y,t) defines a 2D time-varying velocity vector for all points of the 2D space and for an interval of time. The new position of a point is obtained by moving it according to the velocity specified by the vector field at the position of the point during a time interval.

This vector-field based deformation approach is motivated by two observations: First, foldover-free warping is guaranteed, due to the fact that pathlines of vector field do not intersect in the 4D space-time domain, which is a well-known property of vector fields [Von06a]. Second, the deformation is continuous. Since the vector field **v**(x,y,t) is continuous, so is its integral.

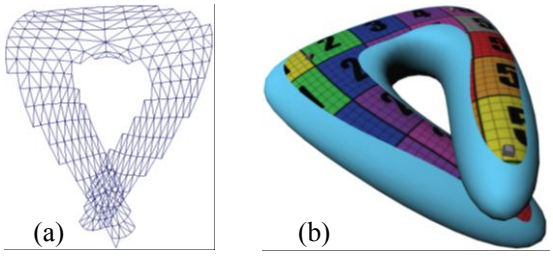


Figure 10. A patch (b) whose planar embedding self-overlaps (a).

One limitation of this method is that it does not allow overlap between different parts of the image region; instead, the method produces highly distorted deformation as shown in Figure 10(b). Allowing overlaps in the warping is necessary in case the planar embedding contains overlapping parts as shown in Figure 10. We show later in this section how to overcome this problem by defining regions of influence for the feature points, allowing the overlapping between different parts of the image region.

5.1.1. Vector Field Based Warping

We warp the image region such that its feature points are moved from their original position (source position) to the position of the corresponding feature points of the patch (target position). The key idea is to relate the movement of the points of the image

region to the trajectory of the feature points moving from the source to the target positions; We define a vector field function **v***(x,y,t)* that relates the instantaneous velocity of the points to those of the feature points along their trajectory. Hereafter, we denote F_i(t) and $\Delta F_i(t)$ the position and velocity of the feature point *i* respectively for $t \in [0,1]$; $F_i(0)$ and $F_i(1)$ represent the source and target positions respectively. The position F_i(t) is obtained from a linear interpolation of $F_i(0)$ and $F_i(1)$.

Given a pair of feature points *i* and *j* and a point *p(t)* in the image region, we compute the relative coordinates $x_{p,i,j}$ and $y_{p,i,j}$ of *p(t)* in the local coordinate frame defined by $F_i(t)$ and $F_j(t)$ (Figure 11):

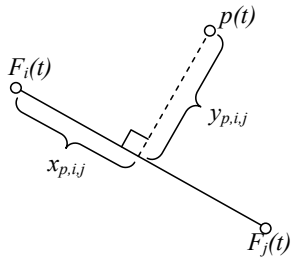


Figure 11. p(t) in the local coordinate frame defined by $F_i(t)$ and $F_j(t)$.

Given the position change of the pair of feature points (*i*, *j*), equation (2) provides the corresponding position change of the point *p(t)*. Note that this equation defines a transformation of *p(t)* which includes rotation, translation and uniform scaling only.

$$p(t) = F_i(t) + x_{p,i,j}\left(F_j(t) - F_i(t)\right) + y_{p,i,j} R_{90}\left(F_j(t) - F_i(t)\right) \quad (2)$$

$$\text{with } R_{90} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Since the number of feature points in the image region is usually more than two, the position of a point is affected by multiple pairs of feature points. Rather than taking into account all possible pairs of

feature point, we use only those pairs whose line segments are completely inside the image region. This guides the deformation to better reflect the shape of the image region.

The total velocity of the point *p(t)* is expressed as a weighted combination of the velocities associated with each pair of feature points. We define a weight for each feature point pair (*i,j*) so that it increases as the point becomes closer to either of the feature points:

$$\beta_{p,i,j} = \frac{1}{\left(d_{p,i} \cdot d_{p,j}\right)^{2\alpha}}$$

$d_{p,i}$ and $d_{p,j}$ are distances between *p(t)* and each of the feature points *i* and *j*. The exponent α determines the smoothness of the warping. We found that the value α=1 gives visually pleasing deformations. These weights are normalized such that their sum is unity for each point *p(t)*.

The function **v**(*x,y,t*) that relates the instantaneous velocity of point *p(t)* with those of the feature points is given by:

$$\mathbf{v}\left(p(t),t\right) = \sum_{i,j}\left(\beta_{p,i,j}\alpha_{p,i,j}\left(\frac{d}{dt}F_i(t) + x_{p,i,j}\left(\frac{d}{dt}F_j(t) - \frac{d}{dt}F_i(t)\right) + y_{p,i,j}R_{90}\left(\frac{d}{dt}F_j(t) - \frac{d}{dt}F_i(t)\right)\right)\right) \quad (3)$$

The purpose of weights $\alpha_{p,i,j}$ is to allow overlaps in warping (see Section 5.1.2). The new position of a point p_0 is obtained by applying a pathline integration of **v**(*x,y,t*) starting form p_0:

$$\begin{cases} \dfrac{dp(t)}{dt} = v\left(p(t),t\right) \text{ for } t \in [0,1] \\ p(0) = p_0 \end{cases}$$

In our current implementation, we use the explicit Euler integration with a constant step size t_s. One integration step involves computing the intermediate solution *p(t)* and updating the values $\beta_{p,i,j}$, $y_{p,i,j}$, and $x_{p,i,j}$ of **v**(*x,y,t*) by using *p(t)* and the feature point positions $F_i(t)$.

5.1.2. Allowing overlaps in the warping:

A distinctive feature of our vector-field based warping is that it deforms the entire 2D space the image region lies in, without regard to the shape of the image region. Consider two distinctive points p_1 and p_2 inside an image region as shown in Figure 11(a). As feature points F_1 and F_4 are designated to become close to each other (Figure 11(b)), the warping produces a highly distorted deformation, which is undesirable. The reason for such artifact is that the positions of the points are affected by all the feature points, regardless of the shape of image region. Allowing overlaps in the image region would be a better alternative, since it reduces such distortion as shown in Figure 12(c).

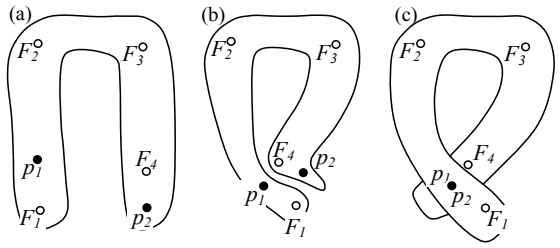


Figure 12. Overlapping of the image region: the image region (a) is deformed such that feature points F_1 and F_2 come close to each other; deformation produced by the vector-field based warping (b); overlapping is acquired by reducing the influence of F_4 on p_1 and F_1 on p_2, respectively.

Overlaps are implemented by restricting the influence of the feature-point pairs to their neighboring area inside the image region. We cluster feature points into groups and divide the image region into segments, each associated with a group. Since the influence of each group of feature points is restricted to a segment, overlap can occur between different segments of the image region.

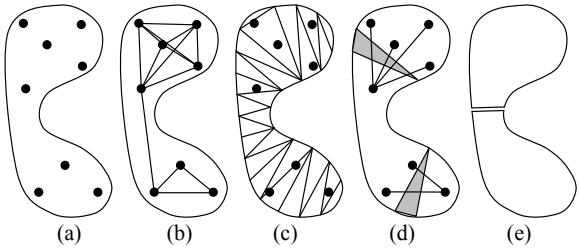


Figure 13. Clustering of the feature point: the boundary of the image region along with feature points provided (a), segments of feature points that do not intersect the boundary (b), constrained Delaunay triangulation (c), clustering of feature points (d), region segments (e).

There are three steps involved in construct the feature-point groups: (1) we first compute the constrained Delaunay triangulation of the image region (Figure 13(c)). (2) We then search for a triangle intersecting with the maximum number of line segments connecting pairs of feature points (Figure 13(d)). (3) The feature points whose line-segment intersects the triangle and which are not already assigned to an existing group are added to a new group. We repeat the process of finding the next triangle with the largest number of intersecting line segments and making a new feature-point group, until every feature point is assigned to a group.

Once all feature points have been assigned to a group, we compute the influence area of each group. For each vertex *p* of the image region, we define a weight $\alpha_{p,l}$ associated with the influence area group *l*

on *p*. We compute these weights individually for each group *l* by minimizing the following function:

$$\min_{\alpha_{p,l}} \sum_{p,q \in E} \left| \alpha_{p,l} - \alpha_{q,l} \right|^2 \qquad (2)$$

Where *E* is the set of adjacent vertices of the constrained Delaunay triangulation such that $p,q \in E$ if either vertex *p* or vertex *q* (or both) are not coincident with a feature point. Coefficient of a feature point is set to either 1 or 0, depending on whether it belongs to group *l* or not.

We use the above computed weights to segment the image region as shown in Figure 13(e); the segmentation is required for the triangulation of the image region (see Section 5.2.1).

Then, for each vertex *p*, we compute the influence weight $\alpha_{p,i,j}$ for each pair of feature points (*i, j*) by summing the weights $\alpha_{p,l}$ of all the groups that contain at least one of the feature points *i* or *j* of the pair:

$$\alpha_{p,i,j} = \sum_{i,j \in l} \alpha_{p,l}$$

These weights $\alpha_{p,i,j}$ are then normalized such that their sum is unity for each vertex *p* of image region. Once computed, these weights are integrated into the equation (3) of the warping. The values $\alpha_{p,i,j}$ continuously change across the vertices of the constrained Delaunay triangulation of the image region, yielding a smooth-looking deformation.

5.1.3. Limitations of the vector field based warping

The warping method does not work in two cases. The first case arises when two feature points belonging to the same pair have same position at the same time during the warping of the image; the local coordinate frame shown in Figure 11 cannot be defined if $F_i(t)$ and $F_j(t)$ are coincident.

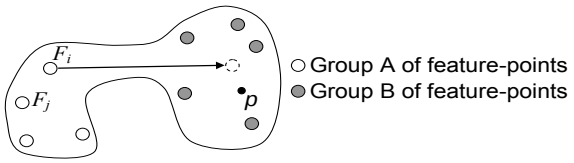


Figure 14. Case for which the warping produces a foldover: a feature point Fi is moved inside the image region to a vertex p whose weigh value $\alpha_{p,i,j}$ is very small.

The second case happens when a pair of feature points (F_i,F_j) is moved inside the image region close to a point *p* whose influence weight $\alpha_{p,i,j}$ is very small.

5.2. Updating the patch structure

As previously stated, we grow the patch iteratively until its planar embedding covers the image region; one iteration consists of computing a parameterization of the patch combined with the warping of the image region, and updating the patch structure. Hereafter, we denote P_{Prev} and P_{Curr} the position vectors of patch vertices corresponding to the parameterization computed at the previous and current iteration respectively. The patch structure is updated as follows: (1) We remove patch triangles that are located outside the image region. Note that although all triangles of the seed patch are initially inside the image region, they may leave the region as we continuously update their parameterization. This is typically encountered when there is a high degree of distortion in the initial mapping of the seed patch, due to the fact that we roughly approximate the initial placement of the seed patch on the surface using geodesic paths without considering its texture mapping distortion. (2) We add triangles that are adjacent to the patch boundary and inside the image region.

In order to obtain a final patch whose surface is topologically equivalent to a disc, we maintain the disc-like topology of the seed patch through all the updates of the patch structure. In particular, we should avoid the patch splitting into more than one as we remove triangles from it. We describe how we avoid such topological change in what follows.

5.2.1. Removing triangles from the patch

In order to maintain the topology of the patch, we test if the intersecting part of the patch with the image region is composed of several disconnected components as illustrated in Figure 15(b) (step (a) in the flowchart of Figure 19). If so, the vertex positions are rolled back to their positions at the previous iteration P_{Prev} where the whole patch was inside the image region (Figure 15(a)). We then compute an approximated parameterization corresponding to the intermediate positions of the vertices between the previous positions P_{Prev} and the positions corresponding to the *final* parameterization (step (b) in Figure 19). The approximated parameterization is computed such that the part of the patch intersecting the image region forms one component (Figure 15(c)).

Rather than computing the exact intersection of the patch with the image region, we find a set *S* of connected vertices from the patch, located inside the image region and containing at least one feature point. If *S* also contains all other feature points, the intersecting part of the patch with the image region can be considered as one component. In order to construct *S*, we first compute a triangulation of the image region by triangulating each of its segments (the algorithm to segment the image region is given in section 5.1.2).

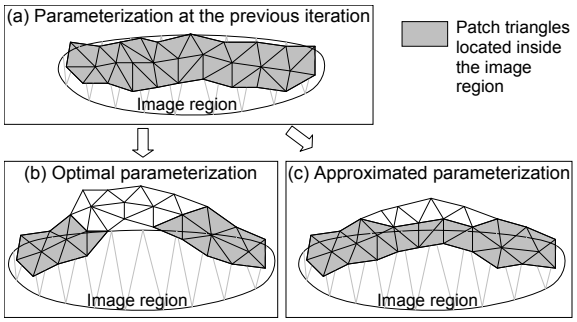


Figure 15. Computing an approximated parameterization.

The algorithm to construct S works by keeping for each vertex v_i of the set S, the triangle T_i of the image region that contains v_i. Initially, S is composed of one feature point; feature points are always inside the image region and we can easily find the triangle of the image region that contains them. The set S is then enlarged iteratively by visiting the edges (v_i,v_j) whose vertex v_i is in the set and v_j outside. Since we know the triangle T_i of the image region that contains v_i, we compute the intersections of (v_i,v_j) with T_i and its neighboring triangles until no more intersecting triangles are found or the edge (v_i,v_j) has crossed the boundary of the image region. If (v_i,v_j) does not intersect the boundary region, v_j is added to S and the triangle T_j that contains v_j is the last triangle intersecting the edge (v_i,v_j). Note that the algorithm to construct S works even when the image region self-overlaps because the test to determine if a patch vertex is inside/outside the image region only requires computing intersections locally with the triangles of the image region.

Once S is constructed, we remove patch triangles whose three vertices do not belong to S (steps (c) and (d) in Figure 19).

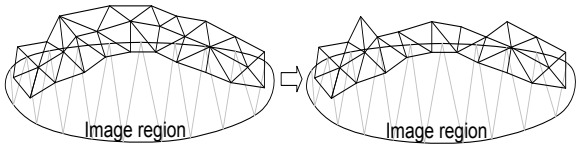


Figure 16. Removing patch triangles whose three vertices are outside the image region.

5.2.2 Insertion of triangles to the patch

There are two different cases when adding triangles to the patch: In the first case, a final parameterization has been computed; we grow the patch by adding a triangle strip along its boundary and inside the image region as shown in Figure 17(a) (step (e) in Figure 19). In the second case, only an approximated parameterization has been calculated; triangles need to be added so that a final parameterization can be computed and the part of the patch intersecting the image region forms one component (step (f) in Figure 19). To achieve this, we find all edges that are shared by two triangles and whose two vertices are on the boundary of the patch (edge e in Figure 17(b)). We then add triangles to the endpoint of the edge which is inside the image region.

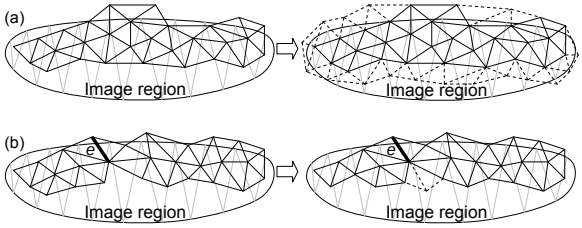


Figure 17. Insertion of triangles.

During the process of adding triangles, we must ensure that the local connectivity of the patch is kept consistent with that of the mesh; given a vertex p of the patch and its corresponding vertex m on the mesh, every vertex connected to p must have one corresponding vertex among those connected to m. The consistency of the local connectivity is broken when two patch vertices corresponding to the same mesh vertex, are adjacent to the same patch vertex (Figure 18(a)). As mentioned in Section 4.2, mesh vertices located inside a self-overlapping texture region (Figure 7) typically have several texture coordinates. To maintain the consistency between the patch and the mesh, we merge the patch vertices corresponding to the same mesh vertex whenever they become adjacent to the same patch vertex (Figure 18(b)).

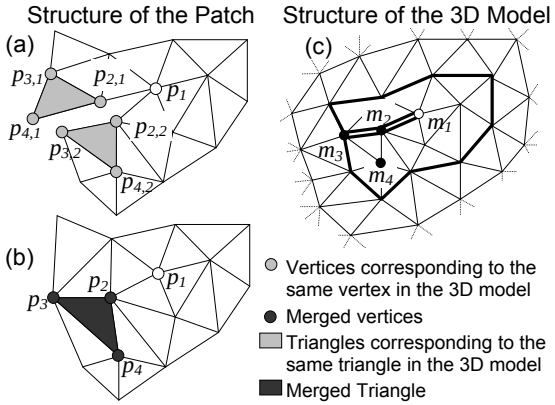


Figure 18. Consider two patch vertices $p_{2,1}$ and $p_{2,2}$ corresponding to the same vertex m_2, and their adjacent vertex p_1 corresponding to m_1. Clearly, the local connectivity around p_1 shown in (a) and the one around m_1 shown in (b) are inconsistent. We merge $p_{2,1}$ and $p_{2,2}$ into p_2 to correct this problem.

We give the flowchart of the algorithm to grow the patch.

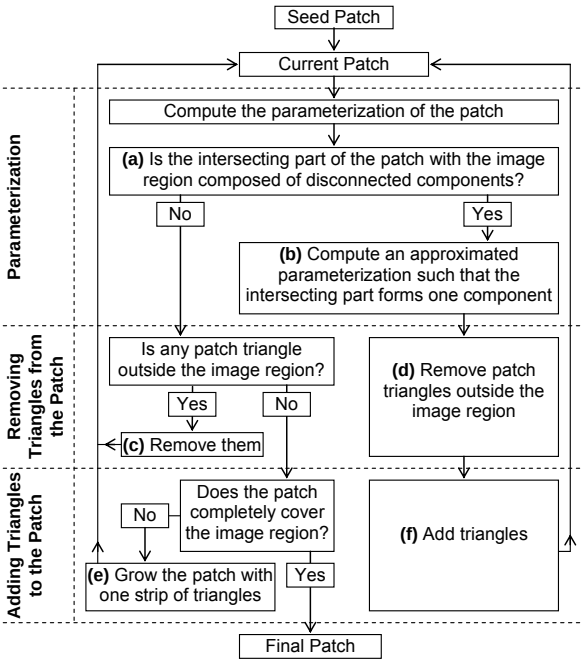


Figure 19. A flowchart of the process of growing the patch.

6. UPDATING THE TEXTURE OF THE 3D MODEL

Once the construction of the patch has been completed, the pixels of image region are transferred into the texture of the 3D model; this implies that the 3D model has already a texture with a global parameterization.

We first warp the image region using vector-field based warping (section 5.1.1) as shown in Figure 20(c). We then update the texture of the 3D model by transferring the image pixels inside the patch triangles into the texture (Figure 20(d)). Note that several images can be mapped successively on the same 3D model to create composition of textures.

Another interesting feature of our approach is to create self-overlapping textures on the 3D model as illustrated in Figure 7. Triangles of the 3D model located in the overlapping parts are textured several times with different portions of the image, each portion corresponding to a triangle in the patch. To determine the hiding-and-hidden relations among the overlapping parts of the texture, we use a technique similar to the painter's algorithm; we sort all the patch triangles by their depths and process them in this order.

The depth value of the patch triangles connected to a feature point is set to an index provided by the artist. Depth values ω_i of other triangles are then computed by minimizing the following function:

$$\min_{\omega} \sum_{(i,j)\in E} \left|\omega_i - \omega_j\right|^2$$

where E is the set of adjacent triangles such that $(i,j) \in E$ if either triangle i or j (or both) are not connected to a feature point.

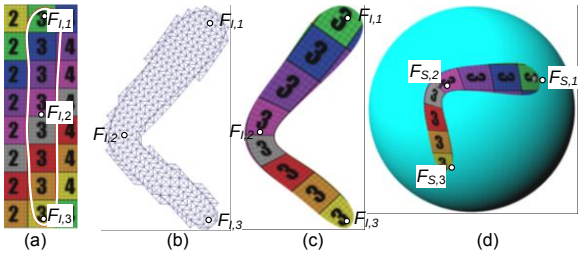


Figure 20. Transferring the pixels of the image region into the texture of the 3D model: image region (a), final patch in the texture space (b), warped image region (c), and 3D model with the updated texture (d).

7. RESULTS

Our texturing tool has been implemented as a plug-in to Maya, with which the user can add and edit feature constraints interactively. The computation time for generating the texture mapping ranges from half a second to a few seconds depending on the complexity of the textured 3D models and the size of the image region.

Our method is demonstrated with several examples corresponding to different cases of texture mapping, showing its versatility. The example shown in Figure 1 and the last one in Figure 23 demonstrate the texture mapping of surfaces with sharp features. The first example in Figure 23 shows how to create overlapping textures on 3D models. The two next examples show the texture mapping with large displacements of the feature points on the 3D model.

7.1. Comparison with previous work

We have compared our method with those proposed by Sun et al. [Sun13a] and Schmidt et al. [Sch06a]. In order to provide a qualitative comparison, we have mapped a same texture onto a same surface using the three different methods (Figure 21). The method by [Sch06a] is clearly the one generating a texture mapping with the highest distortion. [Sun13a] had already mentioned that their method performs better than that of [Sch06a].

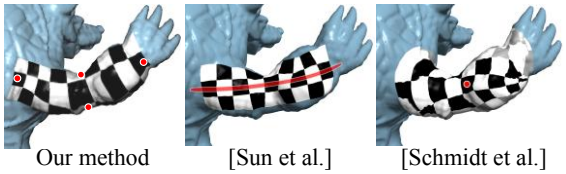


Figure 21. Comparison of our method with the methods proposed by [Sun13a] and by [Sch06a].

Figure 22 shows the texture coordinates generated by our method and by [Sun13a]. In case of [Sun13a], the mapping distortion is high for triangles along the

boundary of the textured area. In case of our method, the mapping distortion is evenly distributed over the textured area.

In order to make a quantitative comparison, we have computed the L^2 stretch metric which measures the deformation of the mapping; the formula of the L^2 stretch is given by [San01a]. As shown in Figure 22, the value of the L^2 metric for our method is significantly lower than that of the method by [Sun13a].

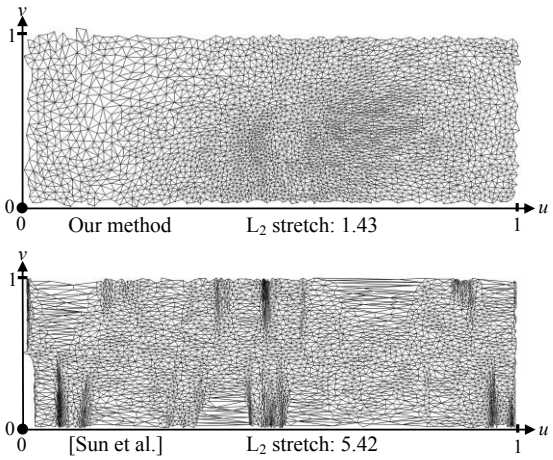


Figure 22. Comparison of the texture coordinates generated by our method with those generated by the method of [Sun13a]. These texture coordinates are those of the surface shown in Figure 21.

7.2. Limitations

Our method has several limitations. As mentioned in section 5.1.3, the proposed warping method may not work when feature points in the warping have same position. Our method fails when the planar embedding of the patch contains triangle flips. However, in practice, we found that triangle flips rarely occur. Finally, since our method is based on conformal parameterization, the textured region tends to grow substantially on surfaces with sharp features; this is because the conformal parameterization minimizes the angular distortion and does not preserve the distances across the surface [Lev02a]. In some cases these undesirable artifacts can be avoided by providing additional feature points. The removal of these artifacts is not always possible. One example is shown in the third row of Figure 23; the level of distortion is so high that it cannot be reduced by placing more feature points.

8. CONCLUSION

We have proposed a method for local parameterization applied for the texture-mapping of images on triangular meshes. Compared to previous work on local parameterization, our method allows multiple feature constraints in the form of correspondence between points in the texture and vertices on the 3D model.

9. REFERENCES

[Car04a] Carr, N. A., and Hart, J. C. 2004. Painting detail. In Proceedings of SIGGRAPH 2004, 842–849.

[Des02a] Desbrun, M., Meyer, M., and Alliez, P. 2002. Intrinsic parameterizations of surface meshes. In Proceedings of Eurographics, 2002.

[Eck01b] Eckstein, I., Surazhsky, V., and Gotsman, C. 2001. Texture mapping with hard constraints. Computer Graphics Forum 20, 3.

[Flo03a] Floater, M., and Hormann, K. 2003. Recent advances in surface parameterization. Multiresolution in Geometric Modelling Workshop.

[Flo03b] Floater, M. 2003. Mean value coordinates. CAGD 20, 1, 19–27.

[Hur99a] Hurtado F., Noy M., and Urrutia J., Flipping edges in triangulations. Discrete Comput. Geom.,22(3):333–346, 1999.

[Kra03a] Kraevoy, V., Sheffer, A., and Gotsman, C. 2003. Matchmaker: constructing constrained texture maps. In Proceedings of SIGGRAPH 2003, 326–333.

[Lef05a] Lefebvre, S., Hornus, S., and Neyret, F. 2005. Texture sprites: Texture elements splatted on surfaces. In ACM SIGGRAPH Symposium on Interactive 3D Graphics (I3D).

[Lev02a] Levy, B., Petitjean, S., Ray, N., and Mallet, J.-L. 2002. Least squares conformal maps for automatic texture atlas generation. In Proceedings of SIGGRAPH 2002, 362–371.

[Lev01b] Levy, B. 2001. Constrained texture mapping for polygonal meshes. In Proceedings of SIGGRAPH 2001, 417–424.

[May05a] Maya, http://www.alias.com/.

[Mey02a] Meyer, M., Lee, H., Barr, A., and Desbrun, M. 2002. Generalized barycentric coordinates on irregular polygons. J. Graph. Tools 7, 1, 13–22.

[Pra00a] Praun, E., Finkelstein, A., and Hoppe, H. 2000. Lapped textures. In Proceedings of SIGGRAPH 2000, 465–470.

[San01a] Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H. 2001. Texture mapping progressive meshes. In Proceedings of SIGGRAPH 2001, 409–416.

[Sch06a] Schmidt R., Grimm C., Wyvill B.: Interactive decal compositing with discrete

exponential maps. In Proceedings of SIGGRAPH 2006, 25(3): 605–613.

[Seo10a] Seo, H., Cordier F.: Constrained Texture Mapping using Image Warping. Comput. Graph. Forum 29(1): 160-174 (2010)

[Sol02a] Soler, C., Cani, M.-P., and Angelidis, A. 2002. Hierarchical pattern mapping. In Proceedings of SIGGRAPH 2002, 673–680.

[Sun13a] Sun Q., Zhang L., Zhang M., Ying X., Xin S.-Q., Xia J., and He Y., Texture brush: an interactive surface texturing interface. In Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '13), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 153-160

[Tur01a] Turk, G. 2001. Texture synthesis on surfaces. In Proceedings of SIGGRAPH 2001, 347–354.

[Von06a] Von Funck W., Theisel H., Seidel H.-P.: Vector field based shape deformations. ACM Trans. Graph. 25(3): 1118–1125 (2006)

[Wei01a] Wei, L., and Levoy, M. 2001. Texture synthesis over arbitrary manifold surfaces. In Proceedings of SIGGRAPH 2001, 355–360.

[Zha05a] Zhang, E., Mischaikow, K., and Turk, G. 2005. Feature-based surface parameterization and texture mapping. ACM Trans. Graphics 24, 1, 1–27.

[Zho04a] Zhou, K., Snyder, J., Guo, B., and Shum, H.-Y. 2004. Iso-charts: Stretchdriven mesh parameterization using spectral analysis. In Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing, 47–56.

[Zho05b] Zhou, K., Wang, X., Tong, Y., Desbrun, M., Guo, B., and Shum, H.-Y. 2005. TextureMontage: Seamless Texturing of Arbitrary Surfaces From Multiple Images. In Proceedings of SIGGRAPH 2005, 1148–1155

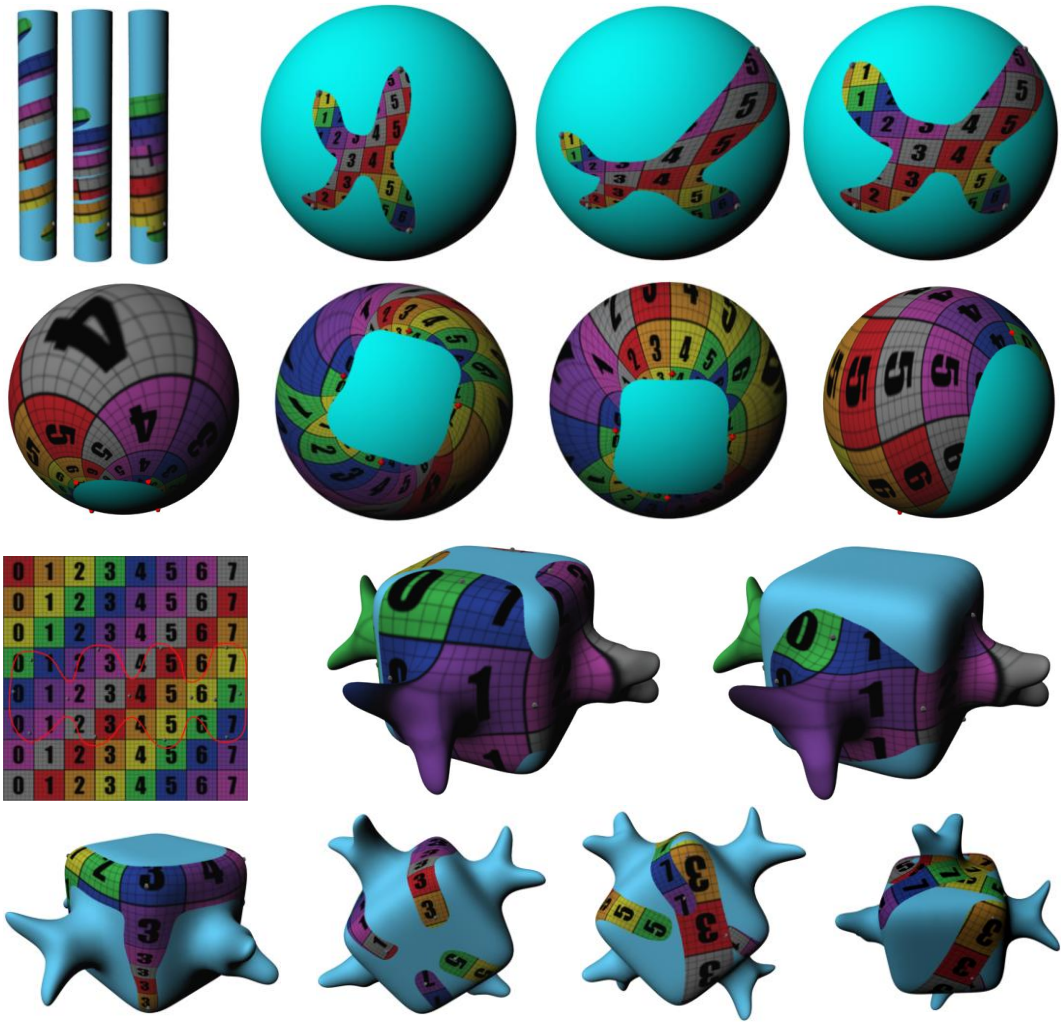


Figure 23. Textured Models.

Rendering of Bézier Surfaces on Handheld Devices

Raquel Concheiro Margarita Amor Emilio J. Padrón
Universidade da Coruña
Facultade de Informática
Campus Elviña, S/N
15071, A Coruña, Spain
rconcheiro | margamor | emilioj @udc.es

Marisa Gil Xavier Martorell
Universitat Politècnica de Catalunya
Campus Nord, Mòdul D6
Jordi Girona, 1–3
08034 BARCELONA, Spain
marisa | xavim @ac.upc.edu

ABSTRACT

Bézier surfaces have been widely employed in the designing of complex scenes with high-quality results. Nevertheless, parametric surfaces cannot be directly rendered in the current GPUs of modern handheld devices. This work proposes a non-adaptive method for tessellating Bézier surfaces on a GPU without primitive generator, such as the GPUs implemented in handled devices. Our technique is based on the utilization of a parametric map of virtual vertices, and its operation can be adapted to the hardware resources available in the GPU by tuning a series of parameters. Additionally, an analysis of the most relevant hardware constraints in the graphics hardware of the current handheld devices has been carried out. As those constraints prevent interactive high-quality results from being achieved, even with our proposal, we present an algorithmic approach focused on the real-time rendering on future handheld devices.

Keywords

Bézier surfaces; GPUs; Handheld Devices; Tuning rendering

1 INTRODUCTION

The market of handheld devices, such as smartphones, consoles or tablets, is nowadays one of the fastest growing technology markets. Graphics processing has become a significant factor on these devices, as consumers' expectations have increased, demanding high quality visual contents and complex render capabilities. Consequently, a new GPU generation has been specifically designed to fit in the constraints of handheld devices: size and power-consumption. Hence, GPUs of these devices implement only a subset of the features available in commodity desktop GPUs. Furthermore, a stripped-down version of the well-known graphics API OpenGL has been developed for these devices: OpenGL ES [Khron10].

Although offline rendering is an important area in computer graphics, especially as far as photorealism is concern, real-time rendering is probably the traditional mainstay of computer graphics. Since an efficiency-quality trade-off is needed in this kind of rendering to maintain interactive rendering rates, the design of an efficient graphics pipeline arises as a key performance factor. This is an issue especially in handheld devices.

Moreover, these rendering pipelines and their supporting graphics hardware are usually designed to work with triangles and vertices. However, these geometric primitives are not always the best option from a modeling point of view. Thus, the use of parametric surfaces to design complex and detailed models has widely spread in fields such as CAD/CAM, virtual reality, animation and visualization. Specifically, the Bézier representation has been widely employed in the designing of high quality complex models [Roger01, Piegl97]. The excellent mathematical and algorithmic properties, combined with successful industrial applications, have contributed to the popularity of this representation.

Bézier surfaces have two significant features from the point of view of a rendering pipeline: compactness, which means low storage and transmission requirements of the resulting models; and scalability, so a surface can be converted into a triangle mesh with few triangles or with many triangles according to the required level of detail (LOD). There are two main approaches for rendering parametric surfaces: tessellation on the CPU or on the GPU. In the first approach, the Bézier surfaces are tessellated into triangles on the CPU, so the resulting triangle mesh is sent down to the GPU to be displayed. This strategy presents some disadvantages that could affect system performance: the amount of information to be transferred from CPU to GPU and the increment in the storage requirements in

the GPU associated with the triangle mesh. These issues are fixed by performing the tessellation directly on the GPU [Guthe05, Dyken09, Conch10, Conch11].

In the case of handheld devices, there are still few proposals dealing with tessellation on the GPU. First works were oriented toward graphics hardware with low programmability, so they were implemented in additional and specific hardware units [Chung09, Chung08]. In [Kim12] a hardware unit for an efficient tessellation in handheld devices was also proposed, but this proposal describes a tessellation procedures for subdivision surfaces.

In this work we present a novel approach to the tessellation of Bézier surfaces on the GPU of handheld devices. Our proposal tessellates parametric surfaces into high-quality triangle meshes that accurately represent complex surfaces and do not contain artifacts such as T-junctions or cracks. It is based on the utilization of a parametric maps of virtual vertices [Boube05, Conch10, Guthe05], what makes it possible to work on GPUs with no primitive generator. More specifically, the guidelines proposed in [Conch10] have had to be adapted to fit the constraints of the graphics hardware in mobile devices, leading us to a completely different implementation, as described in Section 4. Our design allows the efficient exploitation of the information stored in the GPU and the minimization of the CPU-GPU communications. Three main parameters are exposed to allow a fine tuning of the method to the hardware resources available: maximum resolution level, number of surfaces to be rendered per draw call and number of draw calls per frame.

In order to test our approach, we have made an OpenGL ES implementation of the method for Android systems [Goo] and we have designed a full set of experiments to analyze the reasons why Bézier surfaces can not be real-time rendered with good quality by current handheld GPUs. The tests were focused on locating the main performance bottlenecks and identifying possible enhancements and tuning opportunities. Thus, the results obtained could be a useful tool to introduce architecture improvements. Let us emphasize that nowadays, complex triangle meshes can not be rendered in real-time in these devices either [Sarmi12].

This rest of the paper is organized as follows: Section 2 briefly goes over the basics of Bézier surfaces, Section 3 presents our approach to tessellate Bézier surfaces on handheld devices, Section 4 describes the implementation on Android smartphones with OpenGL ES and Section 5 presents the experimental results obtained in our tests. Finally, in Section 6 the main conclusions are highlighted.

2 BÉZIER SURFACES

In this section a brief introduction to the Bézier parametric representation is presented. For reasons of clarity, Bézier curves are first introduced and, after this, the description is extended to Bézier surfaces. An in-depth description can be found in [Piegl97, Roger01].

A Bézier curve is specified by giving a set of coordinate positions, called control points, which indicate the general shape of the curve. These control points are then fitted with piecewise continuous parametric polynomial functions. Mathematically, a parametric n degree Bézier curve is defined by:

$$P(t) = \sum_{i=0}^{n} B_i J_{n,i}(t), \quad 0 \le t \le 1 \tag{1}$$

where B_i are the control points and $J_{n,i}$ are the classical n-degree Bernstein polynomials defined by:

$$J_{n,i}(t) = \binom{n}{i}(1-t)^{(n-i)}t^i \tag{2}$$

where n is the degree of the Bézier basis functions. These functions decide the extent to which a particular control point controls the surface at a particular parametric value t. Only $n+1$ control points and the n-degree Bernstein polynomials are required for the computation of each point of the curve. Note that the first and last control points are coincident with the end points of the curve, that is, $P(0) = B_0$ and $P(1) = B_n$.

The equation for a Bézier curve can be also expressed in matrix form:

$$P(t) = [T][N][G] \tag{3}$$

where $[T] = [t^n \;\; t^{n-1} \ldots \; t^1 \;\; t^0]$, the geometry of the curve is represented as $[G]^T = [B_0 \;\; B_1 \ldots B_n]$, and the $[N]$ matrix is defined by:

$$\begin{bmatrix} \binom{n}{0}\binom{n}{n}(-1)^n & \binom{n}{1}\binom{n-1}{n-1}(-1)^{n-1} & \ldots & \binom{n}{n}\binom{n-n}{n-n}(-1)^0 \\ \ldots & \ldots & \ldots & \ldots \\ \binom{n}{0}\binom{n}{1}(-1)^1 & \binom{n}{1}\binom{n-1}{0}(-1)^0 & \ldots & 0 \\ \binom{n}{0}\binom{n}{0}(-1)^0 & 0 & \ldots & 0 \end{bmatrix}$$

Thus, the matrix form for a cubic Bézier ($n = 3$) is:

$$\begin{aligned} P(t) &= [T][N][G] = \\ &= [t^3 \;\; t^2 \;\; t^1 \;\; 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \end{aligned} \tag{4}$$

Likewise, the shape of a (n,m)-degree Bézier surface is controlled by a set of control points through the equation:

$$Q(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,j} J_{n,i}(u) K_{m,j}(v), \quad 0 \le u,v \le 1 \tag{5}$$

where $J_{n,i}(u)$ and $K_{m,j}(v)$ are the Bézier basis functions in the u and v parametric directions and $B_{i,j}$ are the vertices of a polygonal control net. Again the number of control points in the u and v directions are $n+1$ and $m+1$ respectively. In matrix form, a Bézier surface is given by:

$$Q(u,v) = [U][N][B][M]^T[V] \tag{6}$$

For the specific case of a bicubic Bézier surface, the matrix form is given by:

$$Q(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{7}$$

3 BÉZIER TESSELLATION BASED ON PARAMETRIC MAPS OF VIRTUAL VERTICES

The tessellation of a parametric surface involves the computation of a set of surface points that correspond to the vertices of the triangular mesh, and the identification of the connectivity among them. Since the GPU of current handheld devices do not generate any new geometry, the design of our tessellation proposal is based on virtual vertices [Boube05, Conch10, Guthe05]. This technique uses a parametric map as vertex shader input, with as many positions in the parametric domain as output vertices are needed for the desired resolution of the triangle mesh. Then, by accessing the control points on the Bézier surface to be tessellated, these virtual vertices are evaluated on the vertex shader, generating the resulting triangle mesh. Hence, the resolution of the triangle mesh is chosen by the parametric map being used. Since this approach was initially designed for commodity GPUs, we propose a tuning technique for the effective utilization of the scarce resources available in the GPUs of handheld devices.

Our approach subdivides the parametric domain into uniform squares, where the granularity is selected in function of the desired resolution. More specifically, it tessellates the surface in the parametric space (u,v) in $2^l \times 2^l$ squares of size $\frac{1}{2^l} \times \frac{1}{2^l}$, for a resolution level l that is previously selected by the application taking into account different factors, such as computational power, screen space error or model complexity. Therefore, the Bézier surface is evaluated for each one of the $2^{l+1} \times 2^{l+1}$ to obtain the corresponding Euclidean space points (see Equation 5). The resulting vertices are conveniently arranged to output a triangle strip.

Thus, the grid of parametric values P^l for a resolution level l would be:

$$P^l = \begin{bmatrix} (u_1,v_1) & (u_2,v_1) & \cdots & (u_{2^{l+1}},v_1) \\ (u_1,v_2) & (u_2,v_2) & \cdots & (u_{2^{l+1}},v_2) \\ \vdots & \vdots & \ddots & \vdots \\ (u_1,v_{2^{l+1}}) & (u_2,v_{2^{l+1}}) & \cdots & (u_{2^{l+1}},v_{2^{l+1}}) \end{bmatrix} \tag{8}$$

where

$$u_i, v_i = \frac{i-1}{2^{l+1}-1}, \quad i \in \{1, \cdots, 2^{l+1}\}$$

The base case, $l = 1$, directly projects the control points into the surface to obtain the vertices of the triangle strip.

Before starting, a set of L grids of parametric maps is precomputed on the CPU, where L is the highest resolution level needed: $\{P^1, P^2, \cdots, P^L\}$. These grids are stored in the GPU to be selected and employed as vertex shader input for the different surfaces of the model. The parametric grids are stored in a convenient pattern that implicitly contains connectivity information, preventing the need for any additional indices.

Obviously, the other essential data that need to be accessed by the vertex shader during surface evaluation are the control points. Since memory is a scarce resource in this kind of GPU (the next section explains how and where the Bézier surfaces are stored in the GPU), the surface's data is transferred to the GPU in chunks of N_d Bézier surfaces (of the total N_S surfaces to be rendered for each frame). Therefore, each Draw Primitive call processes a chunk of N_d surfaces, resulting in a total of N_{DP} drawing call for each frame:

$$N_{DP} = \frac{N_S}{N_d}$$

with $1 \leq N_d \leq N_S$.

Thus, if N_d surfaces are processed in each draw call, a total of $N_{samples} = 2^{l+1} \times 2^{l+1} \times N_d$ samples could be concurrently evaluated (assuming a fixed resolution l for all the surfaces in the chunk). This means an input of $N_{samples}$ virtual vertices is needed in the vertex shader, which is provided by N_d copies of the P^l parametric map as vertex shader input (stored in the vertex buffer). Regarding the GPU memory needed for the storage of the N_d Bézier surfaces, the required amount of memory is

$$M = M_{[B^s]} \times N_d \tag{9}$$

where $M_{[B^s]}$ is the storage needed for the control points of each surface and $N_d \ll N_S$ in current handheld devices. Since in most of these devices GPU computation and CPU-GPU transfers do not overlap, each draw call implies a synchronization point, as new M data is sent down to the GPU. The worst case would be a sequential

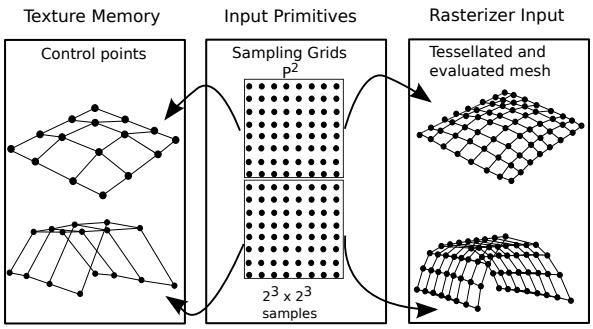


Figure 1: Example of parametric maps for $l = 2$

process of as many draw call as surfaces to render (N_S), with only one surface processed by draw call.

Figure 1 depicts an example of our approach for a resolution level of 2 ($l = 2$) and a couple of Bézier surfaces to be processed concurrently ($N_d = 2$). The parametric map for $l = 2$ is replicated and the resulting samples are the input primitives for the vertex shader (middle box in the figure). The control points of the two surfaces are transferred to the GPU (left box, texture memory is used in this example) and a draw call causes the evaluation of the samples that results in the meshes of the right box.

In summary, GPU performance depends on the right balance between: the number of simultaneous samples $N_{samples}$ that may be concurrently processed, which is a function of N_d and L; the amount of memory needed to storage the N_d Bézier surfaces of a chunk, M; and the number of synchronizations between CPU-GPU, N_{DP}. Therefore, an optimal balance can be expressed by three factors, $\{L, N_d, N_{DP}\}$. Even though two of these three factors are mutually dependent, the analysis is more clear considering the all three.

The number of samples to be processed in parallel may be restricted by the low computational power of the shaders in this kind of GPUs, the size of the vertex buffer or the storage capacity (this is dealt with in the next section). Regarding the influence of each draw call on the performance, it is important to bear in mind that they introduce a certain amount of processing overhead. For each draw call in a OpenGL ES compliant GPU, the graphics driver also collects all current OpenGL ES states, textures and vertex attribute data. The driver processes all this information to generate appropriate commands for the graphics hardware to perform the specified draw operation. This process can take a significant amount of time, and it is even more significant in the case of embedded systems. Finally, to evaluate the number of surfaces that can be processed per draw call, our proposal requires that the control points $[B^S]$ of N_d surfaces be stored in the GPU. Nevertheless, the scarce memory of current handheld devices makes it impossible to store large amount of surfaces.

4 IMPLEMENTATION DETAILS

In this section, we summarize the details of our implementation. The kernel implemented processes bicubic Bézier surfaces and exploits the capabilities of OpenGL ES 2.0. The structure of our algorithm is shown in Figure 2. In the preprocessing stage the grids of virtual vertices P^l, $1 \leq l \leq L$ are transferred from CPU to GPU. During the synthesis process the level of resolution per surface is selected and the control points of N_d surfaces are sent down from CPU to GPU.

As mentioned in the previous section, $N_{samples}$ samples or virtual vertices are sent down to GPU and stored in the *vertex buffer* to compose the input primitives for each vertex shader execution. The control points of each surface $[B^s]$ are stored in a 4×4 float3 arrays $[B^s_x, B^s_y, B^s_z]$.

All draw calls use the same parametric maps, while the resolution level is unchanged, reducing CPU-GPU transfers (i.e. synchronization points). Then, these virtual vertices are evaluated in the vertex shader of the GPU for all the N_d surfaces of a chunk.

Instead of applying the de Casteljau algorithm [Shirl03], in our kernel a direct evaluation strategy is used to compute the tessellation, as it results in a more efficient GPU implementation, avoiding recursion. Figure 3 shows the simple vertex shader pseudocode used for the bicubic surfaces evaluation. The input parameters of the vertex shader (line 1) are the grid parametric values P^l, which are employed in the evaluation of the N_d surfaces of a chunk. The (u, v) parametric values are stored in P^l coordinates x and y whereas the z coordinate stores a surface index $\{0, \cdots, N_d - 1\}$. Thus, each surface within a chunk can be directly indexed (line 8). To evaluate Equation 6 (line 12) $[U]$ and $[V]$ are calculated (lines 9 and 10), the control points of the surface are read from memory (line 11), and the basis functions coefficients (lines 3 to 6) are employed. As a result, the vertices of the final tessellated mesh are obtained.

As will be shown in the results section, the simplicity of this strategy and the efficient management of the data storage are key points, together with the CPU-GPU transfers, for the real time rendering of high quality models.

According to the vertex shader structure of OpenGL ES, this work proposes two different approaches to store Bézier's data in the GPU. The first option, Uniform method, is based on storing the control points of the surfaces in uniform variables, whereas the second one, Texture method, stores them in the texture memory. Both alternatives are described below.

4.1 Uniform method

Uniform variables memory is one type of variable modifiers in the OpenGL ES Shading Language (it has

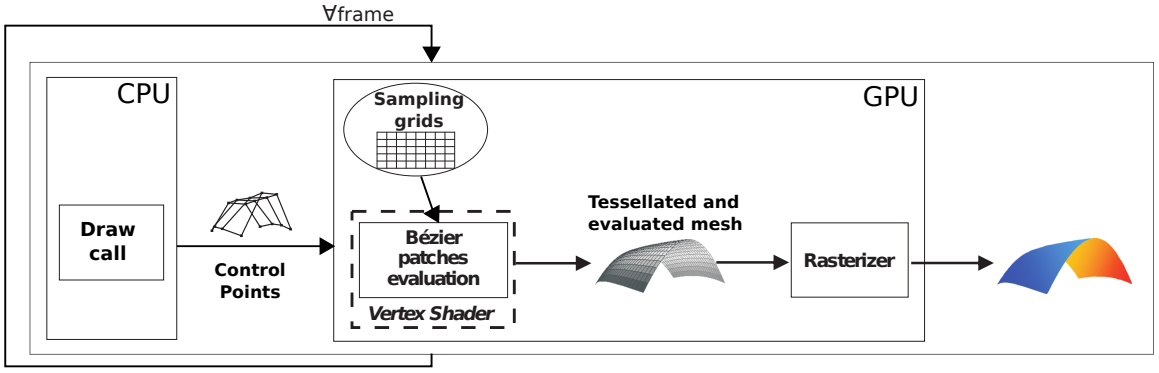


Figure 2: Structure of the method

```
1  VS_OUTPUT DefaultVS(VS_INPUT P^l)
2  {
3    float4x4 [N]= { -1, 3, -3, 1,
4                    3, -6, 3, 0,
5                   -3, 3, 0, 0,
6                    3, 0, 0, 0, }
7    u = P^l.x; v = P^l.y;
8    s = P^l.z × dp × N_d;
9    float1x4 [U]=(u^3, u^2, u, 1);
10   float1x4 [V]=(v^3, v^2, v, 1);
11   float4x4 {[B^s_x],[B^s_y],[B^s_z]} = read from memory (s);
12   float3 vertex = mul([U], [N],[B^s], [N], [V]);
13   return vertex;
14 }
```

Figure 3: Vertex shader pseudocode

evolved in modern desktop GPUs into what is now known as constant memory). These uniform variables are useful for storing all kinds of constant data that shaders can need. Basically, any parameter provided to a shader that is constant across either all vertices or fragments, but that is known before executing the shader should be passed in as a uniform variable. This is the case of the control points of the Bézier surfaces to be tessellated.

From a performance point of view, and according to hardware manufacturers [Mali09, NVIDI11], any access to uniform variable memory is simple and fast and it has a low impact on execution speed. Moreover, this access overhead is similar to an arithmetic operation such as addition or subtraction, and considerably faster than other operation, such as division or square root.

Regarding the capacity of this constant storage, and according to the standard, any implementation of OpenGL ES 2.0 must provide at least uniform memory in the vertex shader, M_{uv}, to store 128 4-float vectors and uniform memory in the fragment shader, M_{uf}, to store 16 4-float vectors. Hence, the maximum number of surfaces per chunk would be

$$N_d = \frac{M_{uv}}{M_{[B^S]}} \tag{10}$$

In the case of bicubic Bézier surfaces, 16 vectors of points are needed to store the control points of each surface, so $N_d = \frac{128}{16} = 8$ is the maximum number of surfaces that can be evaluated in the same draw call, assuming the minimum of uniform variables defined by OpenGL ES 2.0. There are commercial devices that provides a higher number of vertex uniform vectors; for instance Mali 400 provides 256 vertex uniform vectors, $N_d = \frac{256}{16} = 16$.

Clearly, the main drawback of this approach is the reduced number of surfaces that can be stored for each draw call (a low N_d), which means the bottleneck lies in the great number of draw calls needed (a high N_{DP}). This is especially a problem in devices that do not overlap GPU computation and transference.

4.2 Texture method

An alternative to the uniform variables is to store the control points in texture memory, M_T. As texture memory can store a higher number of surfaces than uniform memory, this alternative prevents an important number of draw calls per frame, N_{DP}, one of the main drawbacks of using uniform variables.

$$N_d = \frac{M_T}{M_{[B^S]}} \tag{11}$$

where M_T is considerably larger than M_{uv} and subsequently more primitives can be stored in texture memory than in the uniform variables memory simultaneously. Specifically, for one of our test devices, Mali 400, $M_T = 16MB$, that is four times larger than M_{uv}.

Although Bézier control points can be stored in the texture memory, this storage space has not been designed for store floats. In OpenGL ES 2.0 texture memory formats have been implemented to store color information

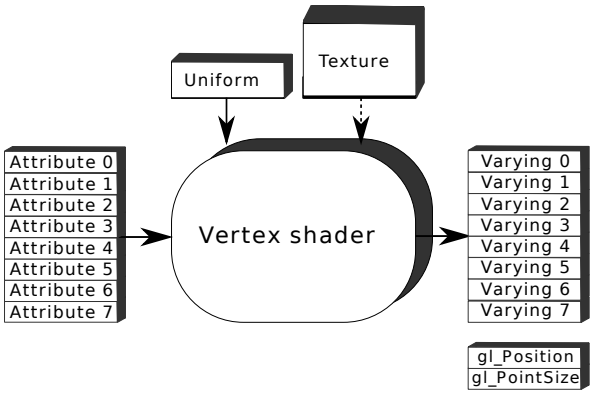


Figure 4: OpenGL ES 2.0 vertex shader

(a) Teacup (b) Teapot

Figure 5: Models employed in the test scenes

(a) L=1 (b) L=3 (c) L=5

Figure 6: Screenshots of the *teacup* model with different levels of resolution

as a 4-byte vector. There are different formats in texture storage, but typically each color is 32 bit data and they are split up into 4 groups of 8 bits: rgba [Foley90] red, green and blue colors and the alpha channel. The texture method defines a codification process to store and recover float values from texture memory. This encoding process to pack a float into a rgba texture is a simple process based on multiplications and divisions by the largest number that can appear.

On the other hand, the texture method solves the main disadvantages of the uniform approach for handheld devices, however it has yet to be implemented on Android platforms. As it is shown in Figure 4 with a dashed line, access to texture memory from vertex shader is not implemented in any commercial OpenGL 2.0 device at this moment. First devices implementing this feature are expected in lately 2013.

5 EXPERIMENTAL RESULTS

In this section, the results of the evaluation of our proposal on different GPU architectures is analyzed. In particular, the platforms we have used are a Samsung Galaxy S2 (*Mali*), a Samsung Galaxy ACE (*Adreno*) and a Asus Transformer TF 300 (*Tegra 3*).

Samsung Galaxy S2 has a 1.2 GHz dual core ARM Cortex-A9 processor and uses ARM's Mali-400 MP GPU with a vertex shader and 4 fragment shaders. Samsung Galaxy ACE features an 800 MHz Qualcomm MSM7227 processor with the Adreno 200 GPU. Adreno 200 GPU implements a unified architecture where a core can dynamically allocate vertex or fragment processing. Finally, Asus Transformer TF300 implements a Nvidia Tegra 3 Quad-core at 1.2GHz and a ULP Geforce 12-core: 4 vertex shaders and 8 fragment shaders.

Different scenes, composed of replicas of a set of models, have been used in our tests. The models (*Teacup* and *Teapot*) are depicted in Figure 5. The number of primitives generated for the different resolution levels is shown in Table 1. Column N_s presents the number of Bézier surfaces whereas the rest of columns include the number of triangles generated for the corresponding level of detail with a uniform tessellation, i.e. all surfaces are tessellated with the same level of detail. Figure 6 depicts a screenshot of the *teacup* model rendered with $L = 1$, $L = 3$ and $L = 5$ in *Tegra 3*. Obviously, a higher level of detail results in a smoother render.

Our analysis is mainly focused on obtaining the optimal tuning factors for the three parameters used to characterize the behavior of our method: $\{L, N_d, N_{DP}\}$. As mention in the previous sections, we consider these three factors in our analysis for clarity reasons, even though two of them are mutually dependent. As the different graphs depict, the results obtained clearly show that our proposal obtains a better performance than the best CPU results: up to 3 fps in *Mali* and 5 fps in *Tegra* for the scene S_{5pots} with $L = 1$. In this CPU implementation each sample is evaluated in the CPU and the whole vertex buffer is sent down to the GPU for each frame.

The first factor we have analyzed in our method is the resolution level, L. Specifically, each of the $2^{l+1} \times 2^{l+1} \times N_d$ samples is evaluated in each GPU vertex shader. The worst configuration was chosen for the other two parameters: $N_d = 1$, so $N_{DP} = N_S$ and there are so many draw calls as surfaces. This configuration is the closest to the CPU tessellation behavior, so the pure computational power determines the difference in performance.

The graphs in Figures 7 and 8 show the frame rate on two distinct platforms for 4 of the test scenes with the different resolution levels. As can be observed, the performance achieved is far better than the described for the CPU, even in this unfavorable configuration. In both devices, *Mali* and *Tegra*, the frame rate drop when the resolution level increases, but some differences between the two platforms can be observed. *Mali* obtains better results when the tessellation level is low (less

Scene	N_s	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$
S_{5cups}	130	2.29	12.44	57.13	244.00	1007.75
S_{5pots}	160	2.81	15.31	70.31	300.31	1240.31
S_{10ups}	260	4.57	24.88	114.26	488.01	2015.51
S_{10pots}	320	5.63	30.63	140.63	600.63	2480.63
S_{15cups}	390	6.86	37.32	171.39	732.01	3023.26
S_{15pots}	480	8.44	45.94	210.94	900.94	3720.94
S_{20cups}	520	9.14	49,77	228.52	976.02	4031.02
S_{20pots}	640	11.25	61.25	281.25	1201.25	4961.25

Table 1: Number of surfaces and triangles generated (in K) for each scene

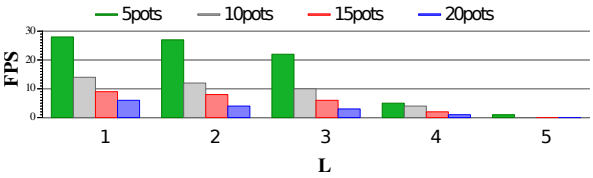


Figure 7: FPS of our implementation in *Mali* with different levels of resolution

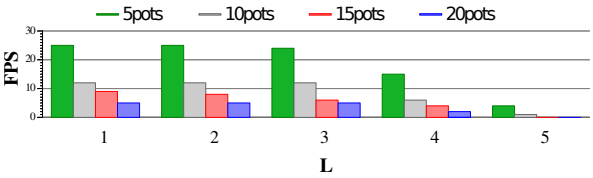
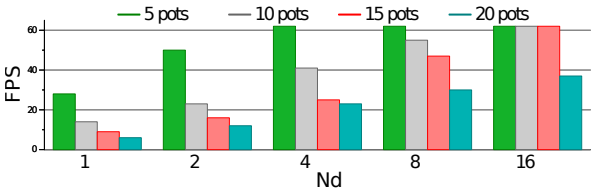
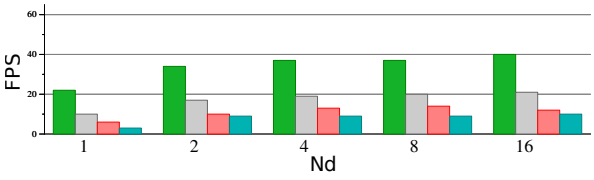


Figure 8: FPS of our implementation in *Tegra* with different levels of resolution



(a) L=1



(b) L=3

Figure 9: *Mali* with different levels of resolution

synchronization penalty) whereas *Tegra* performs better when the resolution level increases (4 vertex shaders vs. 1 vertex shader in *Mali*). In any case, frame rate dramatically drops for $L=5$ in both cases, since a vertex buffer size greater than 16 MB is needed. This information is provided by Table 2, that shows the KBytes stored in the vertex buffer for different scenes and different resolution levels.

In short, the main problems of current GPUs of handheld devices are the computing power and the low number of vertex shaders. This implies a limit on the resolution that can be achieved and the complexity of scenes that can be rendered.

With respect to the rest of factors $\{N_d, N_{DP}\}$ where $N_{DP} = N_S/N_d$, four different scenes have been considered and their performance is depicted with different resolution levels in Figure 9 on the *Mali*. This graph analyzes how the number of surfaces that can be tessellated by a single draw call affects the GPU performance. Similar behavior is observed on *Tegra* and *Adreno* and other scenes. Table3 presents the number of N_{DP} for these scenes with different number of draw calls. As can be observed, values lower than 40 reach maximum performance on *Mali*, that is 62 fps.

Broadly speaking, if the complexity of the model increases (more surfaces, Ns), maintaining a high performance usually implies to try to reduce N_{DP}. For example, S_{5pots} with $N_S = 160$ for $\{L=1, N_d=4, N_{DP}=40\}$ achieves 60 fps, and S_{15pots} with $N_S = 480$ also achieves 60 fps for a configuration $\{L=1, N_d=16, N_{DP}=30\}$. Thus, a trade off between the number of draw calls and the primitives processed in parallel is needed to increase the performance as much as possible.

Figure 10 and Figure 11 present a final comparison of our best results in different platforms: *Adreno*, *Mali* and *Tegra*. Different models such as S_{5pots}, and S_{20pots} have been depicted because a wide range of rendered primitives and vertex buffer size are considered. As has previously been explained, the uniform method stores Bézier control points in the vector uniform variables. Hence, as there are only 128 vector uniform variables in the Adreno 200 architecture, only 8 surfaces can be stored for each draw call, $N_d = 8$, meanwhile up to 16 surfaces can be stored in Mali 400 or in a Tegra 3 in 256 vector uniform variables, $N_d = 16$.

For a low level of resolution, $L=1$ or $L=2$, *Mali* and *Adreno* offer the best performance for $N_d = 8$, that is 62 and 35 fps, respectively. *Tegra* also has the best performance, 58 fps, but scales perfectly as the level of resolution increases, up to 55 fps for $L=3$. Nonethe-

Scene	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$
S_{5cups}	60.94	255.94	1035.94	4155.93	16635.94
S_{5pots}	75.00	315.00	1275.00	5115.00	20475.00
S_{10cups}	121.88	511.88	2071.88	8311.88	33271.88
S_{10pots}	150.00	630.00	2550.00	10230.00	40950.00
S_{15cups}	182.81	767.81	3107.81	12467.81	49907.81
S_{15pots}	225.00	945.00	3825.00	15345.00	61425.00
S_{20cups}	243.75	1023.75	4143.75	16623.75	66543.75
S_{20pots}	300.00	1260.00	5100.00	20460.00	81900.00

Table 2: Vertex Buffer size (in *KB*) for each scene

N_d	S_{5pots}	S_{10pots}	S_{15pots}	S_{20pots}
1	160	320	480	640
2	80	160	240	320
4	40	80	120	160
8	20	40	60	80
16	10	20	30	40

Table 3: N_{DP} for each scene

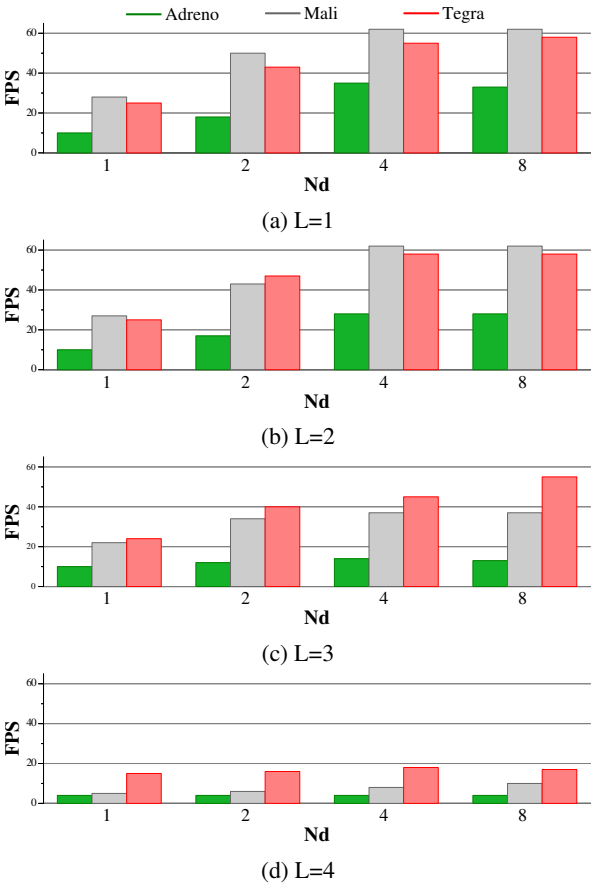


Figure 10: Frame rate comparative in *Adreno*, *Mali* and *Tegra* with S_{5pots} and different resolution levels

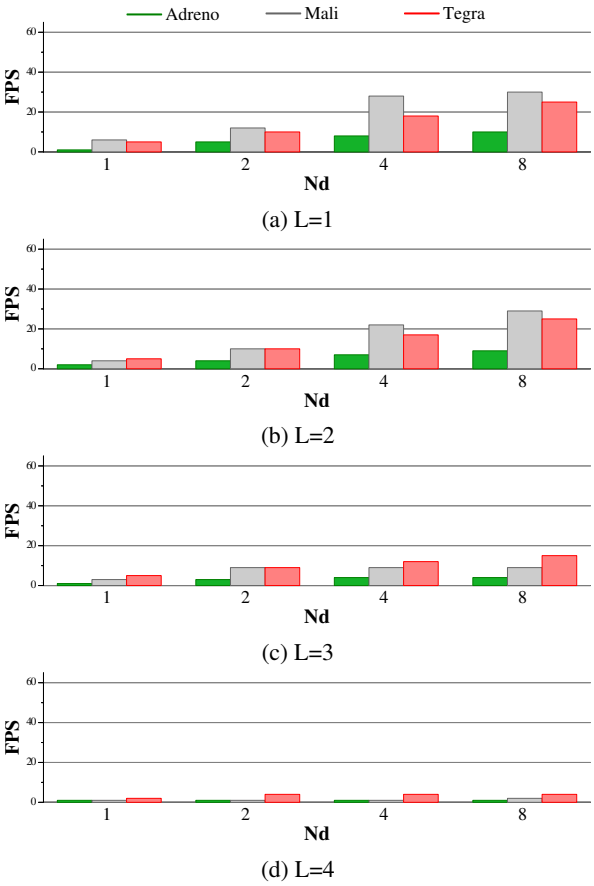


Figure 11: Frame rate comparative in *Adreno, Mali and Tegra* with S_{20pots} and different resolution levels

less, in *Mali* and *Adreno* the performance drop for $L \geq 3$ (37 and 13 fps are achieved, respectively, for $L=3$ and $N_d=8$) confirms that the number of computational cores (i.e. the computational power) becomes a limiting factor and is noticeable in the performance. More specifically, Mali 400 MP GPU has a vertex shader meanwhile Tegra 3 has four. For larger levels of resolution, $L=4$, the performance is below 20 fps for all GPUs due to the low number of vertex shaders.

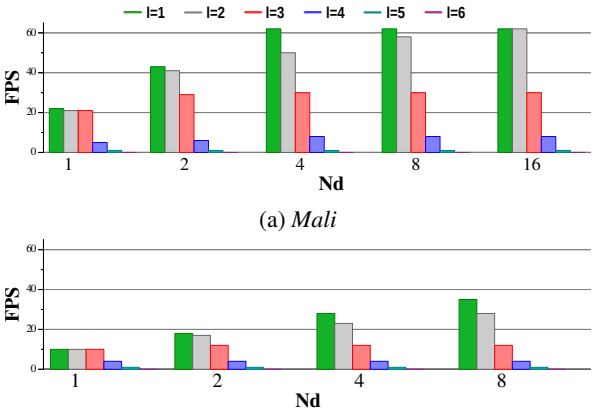
In conclusion, and taking into consideration, for example, the scene S_{20pots} with a setup of $\{L=1, N_d=16, N_{DP}=40\}$ on *Mali*, we achieve only 37 fps, i.e.

a 60% of the maximum frame rate. This performance could be improved with a higher N_d, but that means a greater memory consumption. Besides, with the same scene and platform, but a configuration of $\{L = 1, N_d = 16, N_{DP} = 40\}$, we obtain a poor result of 3 fps due to the lack of computational power. None of these cases allows the use of a realistic illumination algorithm due to the limit in the number of instructions to be executed in the vertex shader [Munsh08]. The maximum number of instructions of the vertex shader across all OpenGL ES 2.0 implementations does not allow the computation of the normals, since most of the instructions are needed to evaluate the surface. On the other hand, as it is commented in [Munsh08], there is no way to query the maximum number of instructions supported by a specific vertex shader.

Nowadays, the Texture method cannot be tested on any market devices, as the access to texture memory from the vertex shader has yet to be implemented in Android platform. This approach would solve the main problem of the uniform method, since texture memory provides a larger storage space than uniform variables. To test how would this approach would be, we have designed a group of tests to measure the texture access latency. Although the evaluation of a Bézier surface cannot be carried out in the fragment shader, the texture memory accesses are processed in this stage to analyze the access latency. Figure 12 shows the performance of a texture memory access from the fragment shader in *Mali* and *Adreno* architectures. A simple model comprising 160 surfaces (S_{5pots}) has been chosen for this test to reduce the impact of computational power and CPU-GPU communication as much as possible. According to the results, the overhead associated with the texture access is about the 20% of the final performance in Mali architecture and under the 10% in an Adreno devices, as a unified architecture as implemented in Adreno devices, dynamically configuring its GPU cores to allocate vertex or fragment processing.

As a result, we can conclude that the proposed texture method could obtain a better performance, as the number of total draw calls could be significantly reduced.

In summary, our analysis shows the main problems of current GPUs in handheld devices, in order to achieve a rendering of Bézier surfaces. These drawbacks are the computing power, the low storage capacity and the low number of vertex shaders and their length. All this implies an important limit on the complexity of the scenes that can be managed, as well as on the realism of the rendering. Thus, simply increasing the device memory would solve the N_d/N_{DP} bottleneck. These constraints should be improved in the future designs of GPUs for these devices. A possible evolution would be a geometry shader-based pipeline, more similar to a DX10/OpenGL 3.2 architecture than to a DX11/OpenGL 4.0. This would allow the exploitation of locality in the generation of new primitives and a greater versatility, in addition to a lower power consumption.



(a) *Mali*

(b) *Adreno*

Figure 12: Performance of scene S_{5pots} with texture access

6 CONCLUSIONS

In this paper we have presented a proposal for the rendering of Bézier surfaces on the GPU of handheld devices. Parametric surfaces cannot be directly rendered in the current GPUs of modern handheld devices, thus our first contribution is to achieve a rendering of Bézier surface. Another related contribution is to describe some of the tuning techniques employed.

Our proposal is based on parametric maps of virtual vertices, and its operation can be adapted to the hardware resources available in the GPU by tuning a series of parameters.

Additionally, an analysis of the most relevant capabilities and constraints in the graphics hardware of the current handheld devices has been carried out by tuning the main parameters of our method. This tuning permits the optimization of the memory usage of the GPU and the minimization of draw calls, that is, the CPU-GPU communication and synchronization barriers.

As a result of our analysis, we can conclude that the current graphics capabilities of these devices are far from allowing the real-time tessellation of complex parametric models. We present our proposal on an algorithmic approach, we do so with an eye toward real-time rendering on future GPUs in handheld devices. As future work, an implementation in a suitable GPU would be worthwhile. Additionally, an adaptive proposal that uses small triangles only where they are needed would better exploit our proposal on GPUs in handheld devices.

ACKNOWLEDGMENTS

This work was carried out through a HIPEAC's Collaboration grant, and all the research has been economically supported by Ministry of Education and Science of Spain under the contracts MEC TIN 2010-16735 and also by the Galician Government under the contracts 'Consolidation of Competitive Research Groups, Xunta de Galicia ref. 2010/6', and CN2012/211, partially supported by FEDER funds.

7 REFERENCES

[Boube05] T. Boubekeur and C. Schlick. Generic Mesh Refinement on GPU. In *Proc. HWWS'05: ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 99–104, New York, NY, USA, 2005. ACM.

[Chung08] K. Chung, C. Yu, D. Kim, and L. Kim. Tessellation-enabled shader for a bandwidth-limited 3D graphics engine. In *Proc. CICC'08: Custom Integrated Circuits Conference*, pages 367 –370, sept. 2008.

[Chung09] K. Chung, C. Yu, D. Kim, and L. Kim. Shader-based tessellation to save memory bandwidth in a mobile multimedia processor. *Computer and Graphics*, 33(5):625–637, 2009.

[Conch10] R. Concheiro, M. Amor, and M. Bóo. Synthesis of Bézier Surfaces on the GPU. In Paul Richard, José; Braz, and Adrian Hilton, editors, *Proc. GRAPP'10: International Conference on Computer Graphics Theory and Applications*, pages 110–115. INSTICC Press, 2010.

[Conch11] R. Concheiro, M. Amor, M. Bóo, and D. Doggett. Dynamic and Adaptive Tessellation of BÃ©zier Surfaces. In *Proc. GRAPP'11: International Conference on Computer Graphics Theory and Applications*, pages 100–105, 2011.

[Dyken09] C. Dyken, M. Reimers, and J. Seland. Semi-uniform Adaptive Patch Tessellation. *Computer Graphics Forum*, 28(8):2255–2263, 2009.

[Foley90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 1990.

[Goo] Google. *Android documentation*.

[Guthe05] M. Guthe, A. Balázs, and R. Klein. GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces. *ACM Transations on Graphics*, 24(3):1016–1023, 2005.

[Khron10] Khronos group. OpenGL ES. Technical report, 2010.

[Kim12] S. Kim, S. Yoon, S. Chung, Y. Kim, H. Kim, K. Chung, and L. Kim. A mobile 3-D display processor with a bandwidth-saving subdivider. *IEEE Trans. VLSI Syst.*, 20(6):1082–1093, 2012.

[Mali09] Mali. Mali GPU OpenGL ES. Application Development Guide. Technical report, 2009.

[Munsh08] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition, 2008.

[NVIDI11] NVIDIA. Technical Brief. Bringing High-End Graphics to Handheld Devices. Technical report, 2011.

[Piegl97] L. Piegl and W. Tiller. *The NURBS Book*. Springer, 1997.

[Roger01] D. F. Rogers. *An Introduction to NURBS with Historical Perspective*. Morgan Kaufmann, 2001.

[Sarmi12] Andrés L. Sarmiento, Margarita Amor, Emilio J. Padrón, Carlos V. Regueiro, Raquel Concheiro, and Pablo Quintía. Evaluating performance of android systems as a platform for augmented reality applications. *International Journal on Advances in Software*, 5(3&4):335–344, 2012.

[Shirl03] P. Shirley. *Fundamentals of Computer Graphics*. Addison-Wesley, 2003.

Creating Finite Element Models of Facial Soft Tissue

Mark Warburton
Department of Computer Science
The University of Sheffield
211 Portobello
Sheffield, S1 4DP, United Kingdom
M.Warburton@dcs.shef.ac.uk

Steve Maddock
Department of Computer Science
The University of Sheffield
211 Portobello
Sheffield, S1 4DP, United Kingdom
S.Maddock@sheffield.ac.uk

ABSTRACT

Physically-based animation techniques enable more realistic and accurate animation to be created. Such approaches require the creation of a complex simulation model that, for computer graphics applications, can efficiently produce realistic-looking animations. We present a process to automatically create animatable non-conforming hexahedral finite element (FE) simulation models of facial soft tissue, including automatic computation of skin layers and element material properties, muscle properties and boundary conditions, making them immediately ready for simulation. Using the GPU, the detailed models can simulate complex gross and fine-scale behaviour, such as wrinkling. Our process can also be used to create a multi-layered FE model of any object (not just soft tissue).

Keywords

physically-based modelling, soft-tissue modelling, facial modelling, physically-based animation, finite element method

1 INTRODUCTION

Facial modelling and animation is one of the most challenging areas of computer graphics. Currently, most facial animation requires performance-capture data, or models to be manipulated by artists. However, using a physically-based approach, the effects of muscle contractions can be propagated through the facial soft tissue to automatically deform the model in a more realistic and anatomical manner.

Physics-based soft-tissue simulation approaches often focus on either efficiently producing realistic-looking animations for computer graphics applications [TW90, KHYS02], or simulating models with high physical accuracy for studying soft-tissue behaviour [BJTM08, KSY08] or surgical simulation [KRG+02, ZHD06]. The former normally simulate large areas, such as the face, using the efficient mass-spring (MS) method [TW90, KHYS02], or physics engines that focus on performance and stability [Fra12]. On the other hand, the latter tend to simulate more detailed models of smaller areas, like a block of skin, using the accurate but computationally complex finite element (FE) method [SNF05, FM08], or the FE-based but precomputation-heavy mass-tensor (MT) method [XLZH11]. Given increases in computational power, and the use of GPU computing architectures, complex FE simulations are now possible in real time [TCO08].

Physics-based simulations require an appropriate simulation model to be created. Such models can either conform to a surface mesh [MBTF03, BJTM08], or a non-conforming model, such as a voxel representation with a bound surface mesh, can be used [DGW11, WM12]. High-quality conforming models that can be efficiently simulated are often difficult and time-consuming to create, and require considerable manual work, although such models may be required for high-accuracy applications. In contrast, non-conforming models can enable more efficient production of stable, realistic-looking animations for computer graphics applications.

The aim of this work is to develop an automatic process to easily create animatable non-conforming hexahedral FE simulation models of facial soft tissue (the soft tissue between the skull and outer skin surface, as shown by Figure 1). In our previous work, basic models were generated that, for example, are unable to simulate anatomical muscle contraction or wrinkles [WM12]. Our current approach can generate much more detailed models that are able to simulate complex gross and fine-scale facial movement, including wrinkles. This approach includes automatic computation of skin layers and material properties, muscle properties, and boundary conditions (such as rigid nodes). The models are

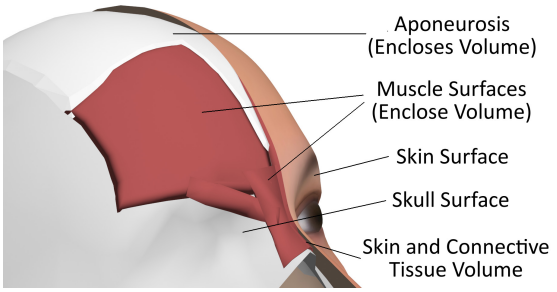


Figure 1: Surfaces and volumes of a facial soft-tissue model. The whole volume between the skull and skin surfaces (i.e. the skin and muscle volumes) is discretised to create an FE facial soft-tissue model.

optimised for GPU simulation, and can be used, for example, to efficiently produce realistic-looking facial animations for computer graphics applications. Our process can also create any multi-layered FE model from any surface mesh (not just soft-tissue models). The following sections detail relevant related work, followed by an overview of our physically-based animation approach, and a description and examples of the model creation process, including model simulation examples using our GPU-based FE system.

2 RELATED WORK

2.1 Physically-Based Facial and Soft-Tissue Models

Physically-based facial animation systems for computer graphics applications normally consist of muscle and skin models, sometimes along with a skull model and wrinkle models. For increased realism, a skull model can include a rotatable mandible [KHYS02]. Muscles have been modelled using vectors [Wat87], and more anatomically accurate geometric [KHYS02] and physics-based volumes [RP07]. Many muscle contraction models are based on a Hill-type model [RP07], some of which are biologically inspired [MHSH10], and the direction of contraction can be approximated as parallel to the central action curve [TZT09], or, more anatomically, by using a fibre field [SNF05].

Various facial soft-tissue models have been proposed, ranging from simple but efficient physics-engine-based [Fra12] and MS models [TW90, KHYS02], to more anatomical and realistic FE models [SNF05, ZHD06]. Detailed models of blocks of skin and soft tissue have also been created [KSY08], along with complex soft-tissue constitutive models [Bis06]. Due to its efficiency, the total Lagrangian explicit dynamic (TLED) formulation has been used for various non-linear FE soft-tissue simulations [TCO08], resulting in large speed-ups. The FE-based MT method has also been used to produce such simulations and also for facial surgical applications [XLZH11], showing similar accuracy to the FE method when simulating small displacements.

2.2 Model Creation Approaches

The model creation process is normally difficult and time-consuming, and it is also dependent on the required model structure. To create an FE model, a suitable simulation mesh must be created, and FE simulation properties, such as boundary conditions, must be set. We focus on simulation meshes constructed using 3D elements to model complex soft-tissue volumes, and these can be conforming or non-conforming (e.g. a voxel representation) with respect to the polygonal surface meshes used for visual purposes. Regarding element types for simulation, we only consider linear elements with a single integration point for optimal computational performance with the high-resolution models.

Simple automatic model creation approaches have been used that just create a layered MS model and skull from a surface mesh [TW90]. On the other hand, CT and MRI scans, or anthropometric data can be used to manually or automatically create an anatomical reference head model [KHYS02]. Such data from the Visible Human Dataset[1] has previously been used for reference model creation [SNF05]. Various techniques have been proposed to deform reference skull, muscle or full physically-based head models using manually defined landmarks [KHYS02, AZ10], although these often rely on good landmark placement. Kähler et al. also developed an interactive editor to enable easy muscle creation by processing user-specified grid points [KHYS02].

Numerous algorithms exist for fast automatic generation of high-quality tetrahedral models that conform to surface meshes [MBTF03, SG05]; however, 4-node tetrahedra are susceptible to volume locking, particularly when simulating incompressible materials like soft tissue. In contrast, reduced-integration 8-node hexahedral elements (with hourglass control) have increased stability and accuracy [WJC+10], particularly when modelling non-linear anisotropic materials [LLT11], and can be used to create meshes using fewer elements, normally outweighing the efficiency of tetrahedra. Hexahedra are therefore often preferred for FE simulations.

Although various algorithms for producing conforming hexahedral meshes have been proposed [SKO+10, ZHB10, NRP11], hexahedral mesh generation is often difficult and time consuming, and, without considerable manual work, many such algorithms suffer problems regarding element quality and robustness, particularly with complex geometries like soft tissue. Techniques have been proposed to improve the quality of hexahe-

[1] http://www.nlm.nih.gov/research/visible/visible_human.html

dral meshes [ISS09, SZM12], although these can produce models with an increased number of elements.

Simple hexahedral meshes can also be merged to produce a complex mesh [SSLS10, Lo12], although these approaches would require a manual decomposition of the complex model such that high quality elements are able to be produced during the merging process. Similarly, conforming and non-conforming domain decomposition FE methods can be used [Lam09], which involve performing an FE analysis on a model decomposed into several independent subdomains. Some other techniques involve deforming a reference hexahedral mesh [CPL00, LLT11], although such approaches require a high-quality reference mesh, and often also require manual work or modifications to the final mesh.

Alternatively, non-conforming hexahedral meshes are easier to create, for example, using voxelisation techniques, and a surface mesh can also be bound to the volume mesh for visual purposes. Such meshes can be used to create models for more stable and computationally efficient FE simulations [DGW11], which can also be optimised for more efficient simulation on the GPU [WM12]. Kumar et al. performed linear elastic FE simulations using structured non-conforming hexahedral grids, and compared these with conforming hexahedral simulation meshes [KPB08], which produced similar stresses, although only relatively simple models were examined. Non-conforming tetrahedral facial and soft-tissue FE models have also been used for stability and performance reasons [SNF05], although linear tetrahedral elements can cause problems such as volume locking.

Once a simulation mesh has been created, model properties, such as element material properties, must be specified. Lee et al. approximated such properties for non-conforming tetrahedral FE soft-tissue models using a sampling procedure [LST09]. A process to generate non-conforming hexahedral FE soft-tissue models has also been proposed [WM12], although this generates very basic models, and neither of these model creation approaches model skin layers (necessary to simulate wrinkles [FM08]), or the sliding of soft tissue over tough deep layers. Techniques have also been proposed to infer muscle fibre directions from B-spline volumes [TSB+05] and conforming volumetric muscle meshes [MHSH10].

Extending existing work [WM12], our model creation process generates more detailed and accurate models, which include skin layers, accurate approximations of element material and muscle properties, and advanced boundary conditions, for example, that can model sliding effects. The models are able to simulate complex gross and fine-scale behaviour, including wrinkling. As well as computer graphics applications, due to the detail and accuracy of the models, our creation process could also be useful in other fields, such as biomechanics and surgery.

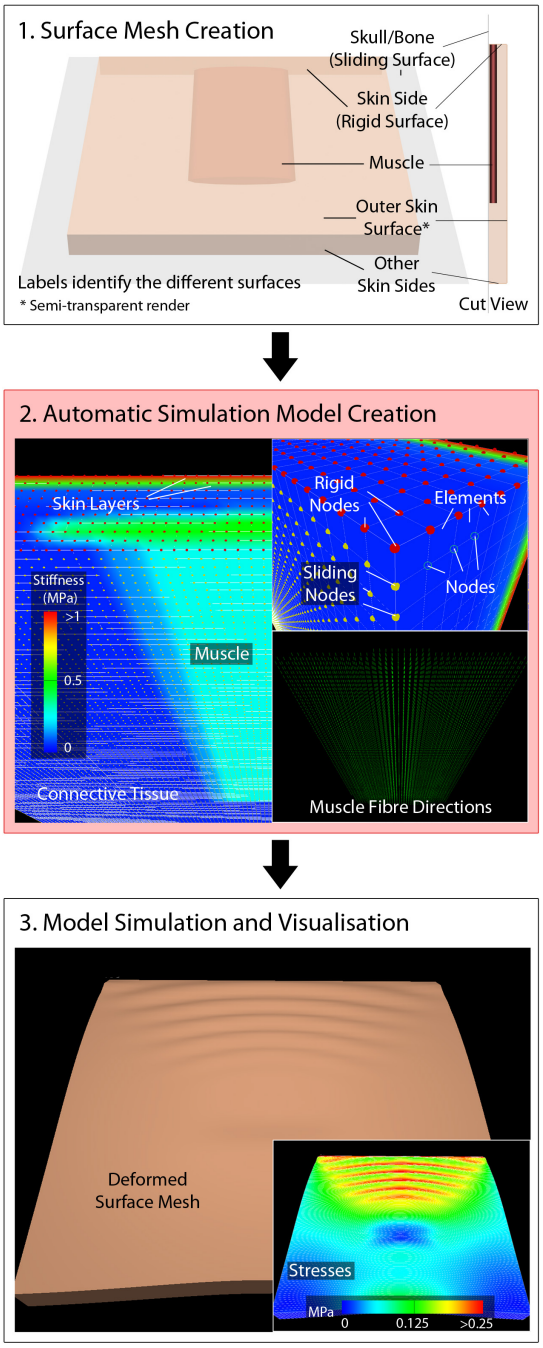


Figure 2: An overview of our physically-based animation approach. Simulation model creation is the focus of this paper.

3 OVERVIEW OF OUR PHYSICALLY-BASED ANIMATION APPROACH

Figure 2 shows an overview of our entire animation approach, which involves three major stages:

1. Creating the surface mesh for an object

2. Creating a suitable simulation model
3. Simulating and visualising the model over time

The surface mesh can be created using any 3D modelling software. The next stage (the focus of this paper) involves using our model creation system to automatically discretise the volumes enclosed by this mesh into a collection of nodes that are connected to form volumetric elements, creating a simulation mesh. FE model parameters are then computed to produce a complete simulation model. We use non-conforming hexahedral models due to model creation, performance and stability advantages [WM12], and surface meshes are bound to these for visual purposes. The models can then be simulated and visualised using a GPU-based FE simulation and visualisation system [WM13].

4 MODEL CREATION

Our model creation process involves five main stages:

1. Voxelising the multi-surface mesh
2. Computing skin layers and element material properties
3. Computing element muscle properties
4. Computing boundary conditions, such as rigid or constrained nodes
5. Binding the surface mesh to the simulation model

The surface mesh can contain various surfaces, including internal surfaces, and volumes are user defined by organising these into closed collections of surfaces. For example, with a facial mesh, there may be a volume for skin and connective tissue (between the skin and skull surfaces), and a volume for each muscle, as shown by Figures 1 and 2. Internal volumes, such as muscles inside the skin volume, overlap the volume they are contained within (i.e. the skin volume doesn't contain holes for the muscles), simplifying surface mesh creation. Volumes can also overlap, for example, to represent the blend of fibres between connecting muscles.

All mesh volumes are grouped into a number of user-defined *levels*, where level 0 is the highest level. Semantically, volumes in a lower level are contained within, and bound by, volumes in the level immediately above. For example, level 0 might consist solely of a skin and connective tissue volume, whereas level 1 might consist of the muscle volumes, which are contained within the skin volume. Other input consists of properties (such as material and muscle properties) associated with each volume, and model properties (such as voxel size). It should be noted that, for a standard soft-tissue model, the volumes and levels are known and can be automatically defined from appropriately labelled surfaces.

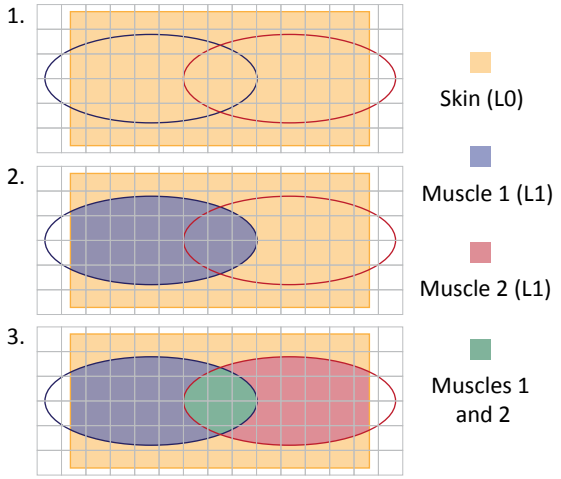


Figure 3: An example of the level-based voxelisation process for a skin block containing two muscles, showing the model state after each mesh volume has been considered in turn. Note this is a 2D illustration of a 3D process.

4.1 Voxelising the Surface Mesh

Starting with a grid of regularly- (cubes) or irregularly-shaped voxels (cuboids) that overlaps the entire surface mesh, voxel properties are calculated based on the proportions of overlap between the voxels and mesh volumes. Enclosed voxels with more than a user-defined proportion of overall overlap (with the union of all level 0 volumes) are used as hexahedral elements.

As illustrated by Figure 3, starting at level 0, sections of voxels that overlap a mesh volume in this level are assigned the properties of that volume. By iteratively considering the next level down, the properties of sections that overlap a mesh volume in both the current and previous levels are overwritten; hence, the properties of sections overlapping the skin volume that also overlap a muscle volume would be overwritten. When a voxel section overlaps multiple mesh volumes in the same level, the properties of these volumes are combined to model the the blend between materials.

From Figure 3, it can be seen that only lenient requirements are imposed on the creation of the surface mesh; for example, as muscles should be contained within the skin volume, parts of muscle surfaces that cross the bounds of this volume are appropriately ignored. Muscle surfaces can therefore simply penetrate the skull, rather than having to attach and conform to the skull surface. This simplifies the modelling of surface meshes by enabling less accurate surfaces to be used without affecting the simulation model.

Similar to some existing approaches for approximating proportions of overlap between element and mesh volumes [LST09], we use a sampling procedure to sample voxels (see Figure 4). Our level-based voxelisation process is performed by assigning the point samples prop-

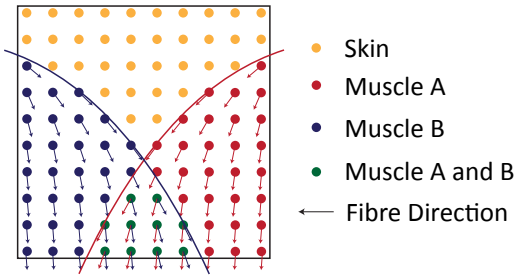


Figure 4: Element samples assigned material and muscle properties, which are used to calculate the overall element properties. Note this is a 2D illustration of a 3D process.

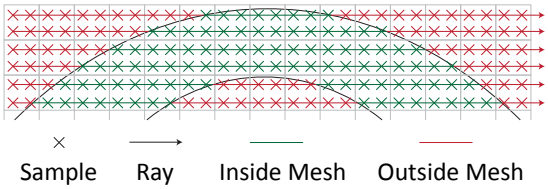


Figure 5: Efficient computation of whether samples are inside a mesh volume using few ray-surface intersections. Note this is a 2D illustration of a 3D process.

erties based on a weighted combination of those associated with level mesh volumes they are contained within:

$$w^{(s,v)} = \begin{cases} \frac{1}{n^{(s)}} & v \text{ encloses } s \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $w^{(s,v)}$ is the weight of mesh volume v with sample s, and $n^{(s)}$ is the number of mesh volumes overlapping the sample. Ray-surface intersection tests determine whether a sample is inside a mesh volume. As shown by Figure 5, by uniformly sampling voxels (rather than, for example, randomly scattered samples [LST09]), a single ray and its surface intersection points can be used to efficently test each sample along an entire line.

4.2 Computing Element Material Properties

Using the voxel element samples, material properties associated with mesh volumes are weighted to calculate element material properties:

$$w^{(e,v)} = \frac{1}{n^{(e)}} \sum_{s=1}^{n^{(e)}} w^{(s,v)} \quad (2)$$

where $w^{(e,v)}$ is the weight of mesh volume v with element e, and $n^{(e)}$ is the number of samples in the element that are enclosed by at least one mesh volume. Within the skin volume, constant thickness layers with different material properties can be generated. Modelling skin layers is necessary to simulate fine details like wrinkles [FM08]. By defining a start depth and

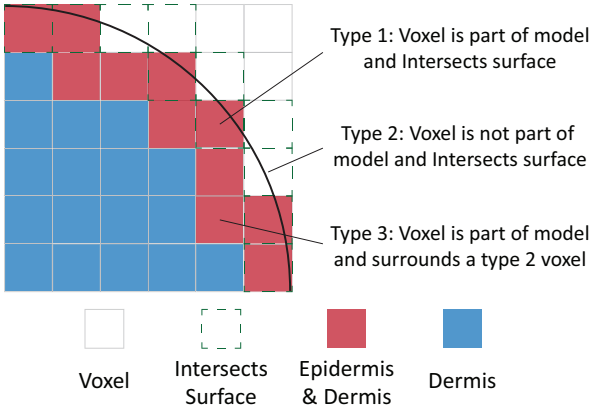


Figure 6: Identification of elements that approximate the outer skin surface and therefore contain epidermal properties. Note this is a 2D illustration of a 3D process.

thickness for each layer, these can also contain overlapping regions, for example, to help capture the non-uniform blend between real skin layers. To determine which layers each sample inside the skin volume is contained within, the distances between the samples and the outer skin surface are tested, and the properties of samples are modified accordingly.

However, without using an extremely high element and sampling resolution, the thin outer epidermal layers are unlikely to be captured using such an approach, and these are therefore treated differently than the other thicker layers. For a single outer epidermal layer, the elements that approximate the outer skin surface are assigned a weight for the epidermal layer of t/e_{avg} (where t is the layer thickness, and e_{avg} is the average element dimension). As shown by Figure 6, such elements are identified as those that are intersected by the outer skin surface, or neighbour such a voxel that isn't included as part of the model (due to insufficient mesh volume overlap).

4.3 Computing Element Muscle Properties

As with material properties, element samples are used to weight muscle stresses for overlapping muscles (see equation 2). As shown by Figure 4, at each element sample, s, a fibre direction, $\mathbf{d}^{(\mathbf{s},\mathbf{m})}$ is also calculated for each muscle, m, that overlaps the sample, and these are averaged to produce a fibre direction, $\mathbf{d}^{(\mathbf{e},\mathbf{m})}$, for each muscle that overlaps the element:

$$\mathbf{d}^{(\mathbf{e},\mathbf{m})} = \frac{1}{n^{(e,m)}} \sum_{s=1}^{n^{(e,m)}} \frac{\mathbf{d}^{(\mathbf{s},\mathbf{m})}}{\|\mathbf{d}^{(\mathbf{s},\mathbf{m})}\|} \quad (3)$$

where $n^{(e,m)}$ is the number of element samples inside the muscle.

Sample fibre directions are calculated using NURBS volume approximations of the muscles, which are

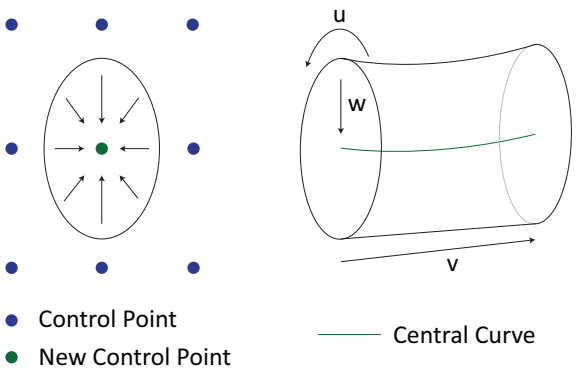


Figure 7: Creation of a NURBS volume by shrinking a NURBS surface, creating a 3rd dimension running from the NURBS surface to a central curve.

created by shrinking NURBS surfaces to their central curve [MLC01], as shown by Figure 7. Such surfaces must be closed along one dimension, and, for each row of control points along this dimension, a new control point is created at the centre of these points to create a central curve. A NURBS volume can then be created, the 3rd parametric dimension of which runs from the NURBS surface to this central curve.

The gradient of the NURBS volume, $V(\mathbf{u})$, with respect to the parametric coordinate along the length of the muscle, a, is used as an implicit fibre field, $d(\mathbf{x})$ [TSB+05]:

$$d(\mathbf{x}) = \frac{\frac{\partial V(V^{-1}(\mathbf{x}))}{\partial a}}{\left\| \frac{\partial V(V^{-1}(\mathbf{x}))}{\partial a} \right\|} \tag{4}$$

$$V(\mathbf{u}) = \sum_{i}^{p} \sum_{j}^{q} \sum_{k}^{r} R_{i,j,k}(\mathbf{u}) \mathbf{c_{i,j,k}} \tag{5}$$

where $\mathbf{u}$ and $\mathbf{x}$ are the parametric and spatial coordinates respectively, $R_{i,j,k}(\mathbf{u})$ are the NURBS volume rational basis functions, $\mathbf{c_{i,j,k}}$ are the control points, p, q and r are the number of control points along u_1, u_2 and u_3 respectively, and $a \in \{u_1, u_2, u_3\}$. As the NURBS volume function requires parametric rather than spatial coordinates, the parametric coordinates of sample points, $V^{-1}(\mathbf{x})$, are estimated using the Newton-Raphson root finding method.

4.4 Computing Boundary Conditions

To define model boundary conditions, using our simulation system, it is possible to restrict the movement of particular nodes by setting them as rigid or sliding (bound by a surface). Rigid nodes remain fixed throughout a simulation, whereas sliding nodes remain a fixed distance from the non-conforming surface they are restricted by, and can be used to model, for example, the sliding effect between the superficial stiff deep facial soft-tissue layers [WMSH10] (a phenomenon often

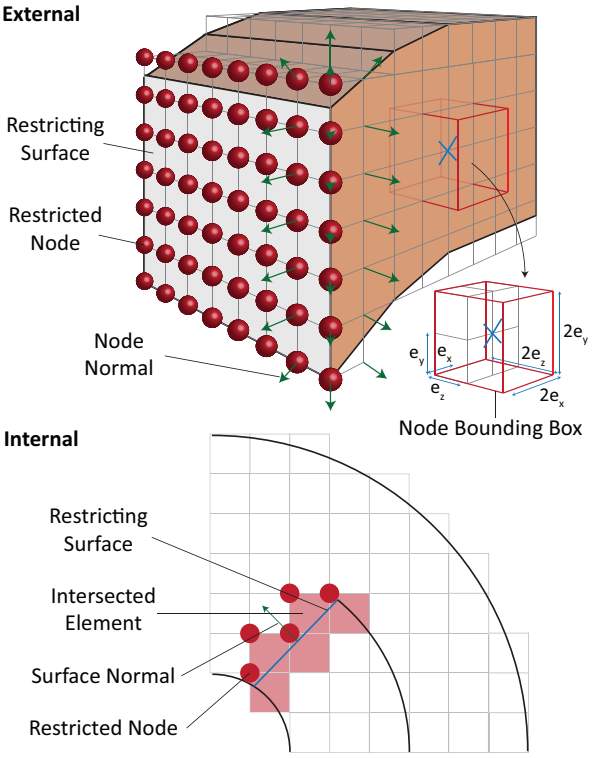


Figure 8: Identification of restricting nodes from external (top) and internal restricting surfaces (bottom, in 2D). In this case, the angle between the external surface and outer node normal must be less than $\pi/2$ for the node to be set as restricted.

neglected with current physics-based facial animations [SNF05, BJTM08]).

As shown by Figure 8, restricted nodes are identified using a collection of non-conforming internal and external rigid and sliding surfaces. An internal surface might represent an attachment area inside the main volume (a level 0 volume), whereas the skull is an obvious external surface that defines part of the boundary of such a volume. With internal surfaces, elements that are intersected by the surfaces are firstly determined. For each node of each such element, if it is located in front of the surface (tested using the closest point on the surface and its normal), it is set as restricted (rigid or sliding).

With external surfaces, only the outermost nodes are considered. To handle cases where a surface lies outside elements without intersecting them, a bounding box with dimensions $2 \times e_i$ (where e_i is the voxel element dimension) is used around each such node centre, and the nodes are recorded if the surface intersects their bounding box. For each such node, if the angle between the node normal and the normal of the closest surface point is less than a specified size (i.e. they are pointing in a similar direction), it is set as restricted. This test is necessary to prevent false positives being detected, for example, if a node lies within range of the surface but is part of an adjoining surface approximation (see Fig-

Layer	Young's Modulus (MPa)	Poisson Ratio	Depth (mm)
Stratum Corneum	48	0.49	0.02
Dermis	0.0814	0.49	1.8
Hypodermis	0.034	0.49	Remains
Muscle	0.5	0.49	Variable
Tendon	24	0.49	Variable

Table 1: The neo-Hookean material properties used for the soft-tissue models.

ure 8). For each sliding node, the signed distance to the surface mesh is also calculated.

4.5 Binding the Surface Mesh, and Model Simulation

For the final stage of the model creation process, as with Dick et al.'s simulation approach [DGW11], the vertices of the surface meshes are bound to and animated with elements of the simulation mesh using trilinear interpolation and extrapolation. A position, $\mathbf{p}$, is bound to the closest element, e (determined by distance tests from the element centres), using three weights, $w_i^{(\mathbf{p})}$ - one along each local axis, i, from the first node of the element, $\mathbf{x}^{(\mathbf{e})}$. For an undeformed voxel element aligned with the global axes, these can be easily calculated:

$$w_i^{(\mathbf{p})} = 1 - \frac{p_i - x_i^{(e)}}{e_i} \tag{6}$$

where e_i is the voxel dimension.

Models generated using our creation process can be simulated using an optimised GPU-based non-linear TLED FE solver [WM13], the full details of which are beyond the scope of this paper. For efficient simulation with this simulation system, the generated models and solver have been optimised to exploit the computational advantages of using such non-conforming hexahedral models on the GPU [WM12]. These optimisations enable both performance and memory advantages resulting from efficient element and node data organisation, facilitating GPU memory coalescing and global memory cache hits, and the reduced amount of element data that is stored, as all elements are initially the same size and shape. This has led to performance increases of almost 2x compared with using a conforming hexahedral simulation model.

5 RESULTS

Figures 9 and 10 show a forehead simulation model that has been generated using our creation process, and the surfaces that were used to generate this. It includes the frontalis, procerus and corrugator supercilli muscles. Figure 2 contains an example of a simpler soft-tissue-block model with a single muscle. As only part of the facial model has been created, nodes along the boundaries to the remainder of the face have been set as rigid to provide the anchoring effect that connecting elements would provide with a full model. While this could cause some artefacts at the model boundaries depending on the size of the external or propagated internal forces at these regions, none are visible in our examples.

Detail	Face	Skin Block	Armadillo
Nodes	629,178	146,410	19,698
Elements	503,530	129,600	15,107
Element Size (mm^3)	0.5^3	0.5^3	2.5^3
Voxel Samples	4^3	4^3	10^3
Model Generation (Single CPU Thread)			
Time (mins:secs)	6:00	0:23	0:41
Memory (MB)	3250	496	148
Simulation (GPU)			
Timestep (ms)	0.005	0.005	0.15
Timestep Computation Time (ms)	13.2	2.9	0.5

Table 2: Statistics of the examples, using an Intel i7-3930K CPU and an NVIDIA GTX 680 GPU.

Tables 1 and 2 show the material properties that were used, and some model and performance statistics. Such complex, high-resolution models are necessary to capture the thin structures, such as skin layers, and simulate fine wrinkling behaviour. As the deep layers are tough and fairly rigid, these are not modelled, and the superficial layers simply slide over the skull or bone surface.

The outer skin surface of the facial mesh was produced using FaceGen[2], while all other polygonal and NURBS surfaces were manually created based on anatomy using 3D modelling tools (see Section 6 for further discussion on surface mesh creation). Both models have constant soft-tissue thickness. As the frontalis has no skull attachment, the galea aponeurotica has been modelled on the forehead model to anchor this muscle and restrict soft-tissue movement towards the top of the head when it contracts. The region of overlap between these structures represents the smooth blend of fibres.

The examples demonstrate the complexity of models that can be created using our creation process, which include skin layers, anatomical fibre directions, and advanced boundary conditions. However, it can be seen that the epidermal layer (stratum corneum) has a higher stiffness and appears thicker than in reality. Due to the low thickness of this layer in relation to element size, the dermis dominates the outer layer of elements. When the material properties of these skin layers are combined for these elements, this results in a stiffness much

[2] http://www.facegen.com/

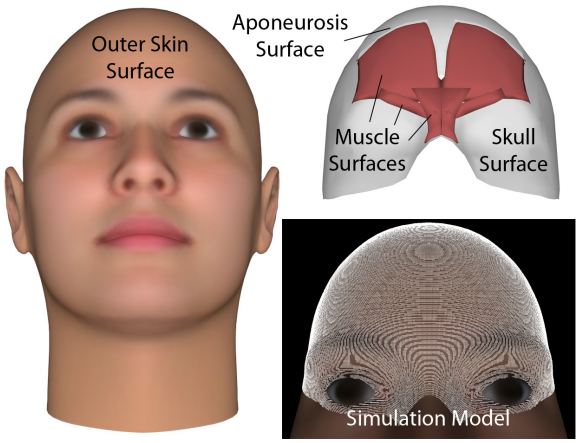


Figure 9: Surfaces and the simulation model for a forehead including the frontalis, procerus and corrugator supercilli muscles.

lower than that of the epidermis alone. The high stiffness is therefore necessary to produce a large enough difference between the stiffness of the two outer layers of elements, which is necessary to simulate wrinkles [FM08]. As a consequence, when using the same material models for the dermis and epidermis, the outer layer of elements acts like a thick epidermal layer. This is unavoidable without using different-shaped elements that can capture the thickness of the epidermal layer.

Regarding performance, the main bottleneck of the model generation was the computation of parametric coordinates of NURBS volumes when computing muscle fibre directions, which used an unoptimised Newton-Raphson root-finding algorithm, and contributed to roughly 38% of the computation time. Also, while memory requirements are quite high, small sections of larger models could be generated independently, as computations of properties for each element and node are independent.

Figures 2 and 11 show animation results using the described example models. More animation results with some parameter variations have been previously presented [WM13]. The elements are treated as 8-node reduced-integration hexahedral elements with hourglass control. Figure 12 shows results of a similar forehead animation using a model with the same number of nodes and elements, but created from a previous simpler version of the model generation process [WM12]. While this model took just 1 minute 20 seconds to generate, using 3.1 GB RAM, and each timestep 9.8ms to simulate (mainly due to no nodal sliding), no wrinkles were produced as only a single skin layer is modelled, and stretching effects can be seen along the lateral edges of the frontalis due to poor approximation of muscle fibre directions. Compression of muscles towards a single point also reduced simulation stability.

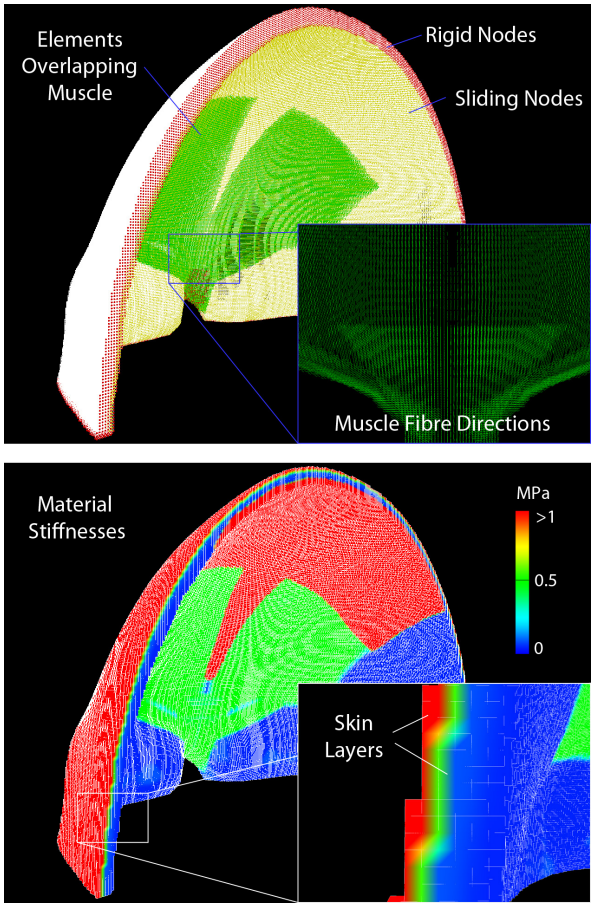


Figure 10: Rear views of the forehead simulation model.

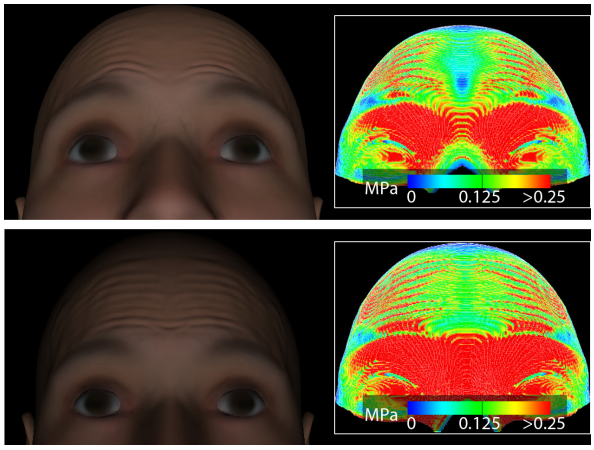


Figure 11: Animation results of forehead models under contraction of the frontalis. The top example uses the model in Figures 9 and 10, whereas the bottom example excludes the procerus and corrugator supercilli muscles, demonstrating the flexibility of the animations by using different models and parameters. Insets show the stresses.

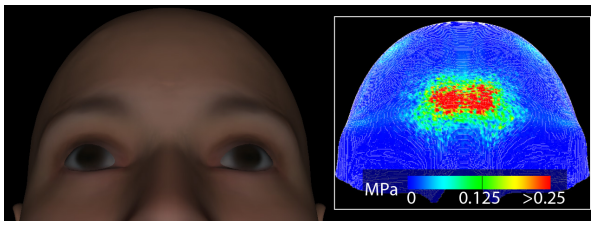


Figure 12: Animation results of a forehead model, created using a previous version of the model generation process [WM12], under contraction of the frontalis.

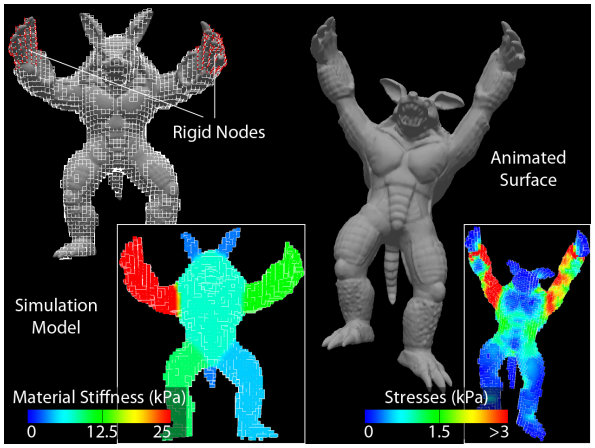


Figure 13: The simulation model and animation results under gravity of a multi-material Stanford Armadillo.

A comparison of using conforming and optimised non-conforming hexahedral models for GPU simulation has been previously discussed [WM12]. Finally, Figure 13 shows an example of a more generic multi-material object, demonstrating the flexibility of our model creation and animation approach. Fewer larger elements were used (but with a higher voxel sampling rate during creation) to animate the gross object movement.

6 DISCUSSION: SURFACE MESH CREATION

For a complex facial model, manually creating and tweaking all the surfaces can be a time-consuming task. This could be made easier, for example, by having an interactive editor for semi-automatic muscle surface creation to simplify muscle definition (like with previous approaches [KHYS02]), although this would probably impact on flexibility and user control. Alternatively, medical data (e.g. CT and MRI data) could be used to create more accurate surfaces. As the simulations are highly dependent on the simulation model and parameters, it is likely that much more realistic soft-tissue animations could be produced by using models generated from such surfaces.

To create a model using medical data, it would first be necessary to use such data to create the separate labelled polygonal surfaces for each structure (including the skull, skin surfaces, muscles and tendons), and the NURBS surface approximations of muscles. The surfaces would need to be hole free due to the sampling procedure that approximates the proportion of overlap between mesh volumes and voxels. However, as most parts of our model creation process, including the sampling procedure, are independent, they could be easily altered or replaced. A different overlap computation technique could therefore be used, which may be more robust to noisy data or volumes with holes, and require less surface mesh clean-up when working with medical data. Once a single reference surface mesh has been created, either manually or from medical data, existing approaches could be used to deform the surfaces of this, such as the muscles, tendons and skull, for easy creation of a new surface mesh for a different face (e.g. based on range scan data) [KHYS02, AZ10].

7 CONCLUSION

This work has presented an automatic process to easily create animatable multi-layered non-conforming hexahedral FE simulation models to which surface meshes are bound, focussing on facial soft-tissue models. Starting with any closed surface mesh, this involves discretising the enclosed volumes into voxels, and calculating model properties (such as skin layers, and element material and muscle properties) based on the proportion of overlap between the volumes and voxels. Boundary conditions are also computed, enabling nodes to be set as rigid (fixed) or sliding (bound by a surface) based on a collection of non-conforming surfaces. The models are optimised with resourceful data storage and organisation for efficient GPU simulation. Examples have demonstrated the complexity and flexibility of models that can be created, and their ability to produce animation of realistic large and fine-scale soft-tissue behaviour.

However, various improvements could be made to the model creation process. Multi-resolution models could be created to enable a higher resolution to be used where necessary, such as along the outer skin surface where wrinkles are produced. Future work will focus on attaching shell elements to the outer skin surfaces to more accurately model the thickness of the outer epidermal layer, and modifying the surface meshes to generate more accurate models. These models will be used with anisotropic viscoelastic materials to produce more realistic animations of different-aged facial movement.

8 REFERENCES

[AZ10] O. O. Aina and J. J. Zhang. Automatic Muscle Generation for Physically-Based Facial Animation. In *SIGGRAPH Posters*, pages 105:1–105:1, 2010.

[Bis06] J. E. Bischoff. Reduced Parameter Formulation for Incorporating Fiber Level Viscoelasticity into Tissue Level Biomechanical Models. *Ann. Biomed. Eng.*, 34(7):1164–1172, 2006.

[BJTM08] G. Barbarino, M. Jabareen, J. Trzewik, and E. Mazza. Physically Based Finite Element Model of the Face. In *Proc. ISBMS*, pages 1–10, 2008.

[CPL00] B. Couteau, Y. Payan, and S. Lavallée. The mesh-matching algorithm: an automatic 3D mesh generator for finite element structures. *J. Biomech.*, 33(8):1005–1009, 2000.

[DGW11] C. Dick, J. Georgii, and R. Westermann. A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simul. Model. Pract. Theory*, 19(2):801–816, 2011.

[FM08] C. Flynn and B. A. O. McCormack. Finite element modelling of forearm skin wrinkling. *Skin Res. Technol.*, 14(3):261–269, 2008.

[Fra12] M. Fratarcangeli. Position-based facial animation synthesis. *Comput. Animat. Virtual Worlds*, 23(3-4):457–466, 2012.

[ISS09] Y. Ito, A. M. Shih, and B. K. Soni. Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates. *Int. J. Numer. Methods Eng.*, 77(13):1809–1833, 2009.

[KHYS02] K. Kähler, J. Haber, H. Yamauchi, and H.-P. Seidel. Head shop: Generating animated head models with anatomical structure. In *Proc. SCA*, pages 55–63, 2002.

[KPB08] A. V. Kumar, S. Padmanabhan, and R. Burla. Implicit boundary method for finite element analysis using non-conforming mesh or grid. *Int. J. Numer. Methods Eng.*, 74(9):1421–1447, 2008.

[KRG+02] R. M. Koch, S. H. M. Roth, M. H. Gross, A. P. Zimmermann, and H. F. Sailer. A Framework for Facial Surgery Simulation. In *Proc. SCCG*, pages 33–42, 2002.

[KSY08] O. Kuwazuru, J. Saothong, and N. Yoshikawa. Mechanical approach to aging and wrinkling of human facial skin based on the multistage buckling theory. *Med. Eng. & Phys.*, 30(4):516–522, 2008.

[Lam09] B. P. Lamichhane. Mortar Finite Elements for Coupling Compressible and Nearly Incompressible Materials in Elasticity. *Int. J. Num. Anal. Model.*, 6(2):177–192, 2009.

[LLT11] M.-F. Li, S.-H. Liao, and R.-F. Tong. Facial hexahedral mesh transferring by volumetric mapping based on harmonic fields. *Comput. Graph.*, 35(1):92–98, 2011.

[Lo12] S. H. Lo. Automatic merging of hexahedral meshes. *Finite Elem. Anal. Des.*, 55:7–22, 2012.

[LST09] S.-H. Lee, E. Sifakis, and D. Terzopoulos. Comprehensive Biomechanical Modeling and Simulation of the Upper Body. *ACM Trans. Graph.*, 28(4):99:1–99:17, 2009.

[MBTF03] N. Molino, R. Bridson, J. Teran, and R. Fedkiw. A Crystalline, Red Green Strategy for Meshing Highly Deformable Objects with Tetrahedra. In *Proc. IMR12*, pages 103–114, 2003.

[MHSH10] K. Mithraratne, A. Hung, M. Sagar, and P. J. Hunter. An Efficient Heterogeneous Continuum Model to Simulate Active Contraction of Facial Soft Tissue Structures. In *Proc. WCB*, pages 1024–1027, 2010.

[MLC01] D. Ma, F. Lin, and C. K. Chua. Rapid Prototyping Applications in Medicine. Part 1: NURBS-Based Volume Modelling. *Int. J. Adv. Manuf. Technol.*, 18(2):103–117, 2001.

[NRP11] M. Nieser, U. Reitebuch, and K. Polthier. CUBECOVER – Parameterization of 3D Volumes. *Comp. Graph. Forum*, 30(5):1397–1406, 2011.

[RP07] O. Röhrle and A. J. Pullan. Three-dimensional finite element modelling of muscle forces during mastication. *J. Biomech.*, 40(15):3363–3372, 2007.

[SG05] H. Si and K. Gärtner. Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations. In *Proc. IMR14*, pages 147–163, 2005.

[SKO+10] M. L. Staten, R. A. Kerr, S. J. Owen, T. D. Blacker, M. Stupazzini, and K. Shimada. Unconstrained plastering – Hexahedral mesh generation via advancing-front geometry decomposition. *Int. J. Numer. Methods Eng.*, 81(2):135–171, 2010.

[SNF05] E. Sifakis, I. Neverov, and R. Fedkiw. Automatic Determination of Facial Muscle Activations from Sparse Motion Capture Marker Data. *ACM Trans. Graph.*, 24(3):417–425, 2005.

[SSLS10] M. L. Staten, J. F. Shepherd, F. Ledoux, and K. Shimada. Hexahedral Mesh Matching: Converting non-conforming hexahedral-to-hexahedral interfaces into conforming interfaces. *Int. J. Numer. Methods Eng.*, 82(12):1475–1509, 2010.

[SZM12] L. Sun, G. Zhao, and X. Ma. Quality improvement methods for hexahedral element meshes adaptively generated using grid-based algorithm. *Int. J. Numer. Methods Eng.*, 89(6):726–761, 2012.

[TCO08] Z. A. Taylor, M. Cheng, and S. Ourselin. High-Speed Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units. *IEEE Trans. Med. Imaging*, 27(5):650–663, 2008.

[TSB+05] J. Teran, E. Sifakis, S. S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw. Creating and Simulating Skeletal Muscle from the Visible Human Data Set. *IEEE Trans. Vis. Comput. Graph*, 11(3):317–328, 2005.

[TW90] D. Terzopoulos and K. Waters. Physically-Based Facial Modeling, Analysis, and Animation. *J. Vis. Comput. Animat.*, 1(2):73–80, 1990.

[TZT09] C. Y. Tang, G. Zhang, and C. P. Tsui. A 3D skeletal muscle model coupled with active contraction of muscle fibres and hyperelastic behaviour. *J. Biomech.*, 42:865–872, 2009.

[Wat87] K. Waters. A muscle model for animation three-dimensional facial expression. In *Proc. SIGGRAPH*, pages 17–24, 1987.

[WJC+10] A. Wittek, G. Joldes, M. Couton, S. K. Warfield, and K. Miller. Patient-specific non-linear finite element modelling for predicting soft organ deformation in real-time; Application to non-rigid neuroimage registration. *Prog. Biophys. Mol. Biol.*, 103(2–3):292–303, 2010.

[WM12] M. Warburton and S. Maddock. Creating Animatable Non-Conforming Hexahedral Finite Element Facial Soft-Tissue Models for GPU Simulation. In *Proc. WSCG*, pages 317–325, 2012.

[WM13] M. Warburton and S. Maddock. Physically-Based Forehead Animation including Wrinkles. In *Proc. CASA*, 2013.

[WMSH10] T. Wu, K. Mithraratne, M. Sagar, and P. J. Hunter. Characterizing Facial Tissue Sliding Using Ultrasonography. In *Proc. WCB*, pages 1566–1569, 2010.

[XLZH11] S. Xu, X. P. Liu, H. Zhang, and L. Hu. A Nonlinear Viscoelastic Tensor-Mass Visual Model for Surgery Simulation. *IEEE Trans. Instrum. Meas.*, 60(1):14–20, 2011.

[ZHB10] Y. Zhang, T. J. R. Hughes, and C. L. Bajaj. An Automatic 3D Mesh Generation Method for Domains with Multiple Materials. *Comput. Methods Appl. Mech. Eng.*, 199(5–8):405–415, 2010.

[ZHD06] S. Zachow, H.-C. Hege, and P. Deuflhard. Computer-Assisted Planning in Cranio-Maxillofacial Surgery. *J. Comp. Inf. Technol.*, 14(1):53–64, 2006.

Visual Parameter Exploration in GPU Shader Space

Peter Mindek
Vienna University of Technology
Austria
mindek@cg.tuwien.ac.at

Stefan Bruckner
University of Bergen
Norway
stefan.bruckner@uib.no

Peter Rautek
King Abdullah University of Science and Technology
Saudi Arabia
peter.rautek@kaust.edu.sa

M. Eduard Gröller
Vienna University of Technology
Austria
groeller@cg.tuwien.ac.at

ABSTRACT

The wide availability of high-performance GPUs has made the use of shader programs in visualization ubiquitous. Understanding shaders is a challenging task. Frequently it is difficult to mentally reconstruct the nature and types of transformations applied to the underlying data during the visualization process. We propose a method for the visual analysis of GPU shaders, which allows the flexible exploration and investigation of algorithms, parameters, and their effects. We introduce a method for extracting feature vectors composed of several attributes of the shader, as well as a direct manipulation interface for assigning semantics to them. The user interactively classifies pixels of images which are rendered with the investigated shader. The two resulting classes, a positive class and a negative one, are employed to steer the visualization. Based on this information, we can extract a wide variety of additional attributes and visualize their relation to this classification. Our system allows an interactive exploration of shader space and we demonstrate its utility for several different applications.

Keywords

parameter space exploration, shader augmentation

1 INTRODUCTION

In data visualization, GPU shader programs are often used to process large amounts of data and to create suitable visual representations. In this case, the shaders usually implement algorithms which provide a mapping between the data and the intended visual representation. The way how the data is displayed depends on the shader program and its input parameters. The vector of values of the input parameters is a point in the corresponding parameter space. An interpretation of the resulting image requires knowledge of the relationships between the parameter space of the underlying algorithm and the visualization. These relationships might not be trivial to grasp given only the source code of the shader implementing the algorithm and the resulting image.

Data visualization algorithms implemented as GPU shaders can be investigated from various perspectives. Rendered images (i.e., image space), are presented to the user. The mapping process from data space to image space is specified in a shader program. We refer to the combination of possible shader programs and their parameters as shader space. While the image space is usually interesting for the domain expert for whom the visualization mapping was originally created, the shader space is interesting for the visualization expert. In situations where the shader program needs to be modified, it is important for the visualization expert to understand how the algorithm affects the resulting visualization. Therefore, there is a necessity for tools allowing for exploration of the shader space.

We propose a method that allows users to explore GPU shader programs and their parameter spaces by assigning semantic classifications to parts of the rendered images. This user-assigned information is subsequently used to modify the visualization mapping by generalizing it to the whole rendered image. In this way, the influence of the examined parameters or data attributes on the display of the features of interests becomes apparent.

The shader exploration is facilitated by a well defined domain specific language (DSL) that extends the shader language under consideration. The DSL is used to annotate the shader program without changing its functionality. The annotations are inserted as comments that are parsed and interpreted by our system. This strategy is similar to well known documentation tools like Doxygen and JavaDoc. The benefits are:

- the annotations are kept close to the original code (resulting in easier maintainability)

- the annotations are transparent to the host language.

We refer to the insertion of comments that include commands from the DSL as shader augmentation. The augmented shader can either be run unmodified to fulfil its original purpose, or be transformed for the shader space exploration. This provides functionality for the exploration of desired parameters and data attributes. As the additional functionality is part of the augmented shader, it is executed on the GPU, which benefits the performance of the system.

The main contributions of this paper are:

- a method for translating semantic classifications from image space to shader space and using them to explore the effects of parameters on the rendered images
- a parameter exploration language (PEL), which is used to extract values of relevant parameters and data attributes from the shader program for further parameter-space exploration
- a graphical user interface for the automatic and transparent generation of PEL code.

While the PEL can be employed by visualization experts, the graphical user interface allows the shader-space exploration even without knowledge of the shading language, which might be a benefit for domain experts.

2 RELATED WORK

Our method is designed for exploring parameter spaces of various visualization algorithms. The method is particularly applicable to volume rendering. Volume rendering algorithms are usually complex and it is difficult to predict the effects of their parameters and data attributes. In this work, we focus mainly on volume rendering as a possible application area for our method.

In volume rendering, the mapping between data and rendered images is usually adjusted using transfer functions. A transfer function maps one or more data attributes to optical properties. It enables users to visually examine these attributes of the data. Wu and Qu [20] present a framework for interactive transfer function design based on genetic algorithms and image similarity, where the user edits direct volume rendered images. Another approach is presented by de Moura Pinto and Freitas [3]. It is based on dimensional reduction of the voxel attributes using self-organizing maps. Correa and Ma [1] propose visibility-driven transfer functions for enhancing the visibility of important features in volume data. Tzeng and Ma [19] propose an interface for data classification in a cluster space of different materials present in the data.

Tzeng et al. [18] propose a volume data classification approach based on machine learning. This approach employs a sketch-based interface to select a primary classification, which is then used as a training set for a machine learning algorithm. Similar work is also presented by Guo et al. [7]. Their WYSIWYG (What You See Is What You Get) approach allows definition of one-dimensional transfer functions by direct interaction with the rendered images. Yuan et al. [21] present a method for sketch-based volume segmentation. Visualization mapping based on semantics is proposed by Rautek et al. [14]. In this method, the mapping of the data attributes to visual styles are described by domain experts in the natural language. An extension of this work [15] allows to use interaction-dependent rules for specifying the visualization mapping.

Gavrilescu et al. [5] present a work on user interfaces offering information on the effects of parameters that are adjusted by these interfaces. Our method also enables to explore the effects of the parameters. However, we provide a possibility to observe the effects on specified features in the visualization. This makes our method useful not only in data exploration, but also in the exploration of the visualization algorithms and their behaviour.

The visualization-algorithm analysis capabilities of our method can be used for debugging purposes as well. Various systems for analyzing and debugging visualization software have been presented [4] [8] [16]. Crossno and Angel [2] present a case study on debugging of visualization software. Meyer-Spradow et al. [13] propose a framework for rapid prototyping of visualization algorithms by connecting modules into a data-flow network. The framework also provides debugging abilities by displaying outputs of the individual modules. Our method allows to visually identify effects of particular shader parameters and data attributes on the rendered images. Rather than displaying values of individual variables, it provides visual feedback on how the values correspond to the user-defined classification of the rendered image. These findings can be compared to expected behaviours for debugging purposes or the visualization-algorithm analysis. Our method can be used as a complementary tool to existing shader debugging solutions.

Our work is related to data-exploration methods, since it uses similar principles to explore the shader space of the visualization algorithms. Jankun-Kelly et al. [10] propose a model for the visualization exploration process. McCormick et al. [11] propose Scout - a visualization system utilizing the GPU for exploration and visualization by applying queries directly to the data. Gerl et al. [6] propose a method where properties of visualization mappings are rendered, and brushing is used on these rendered images to explicitly spec-

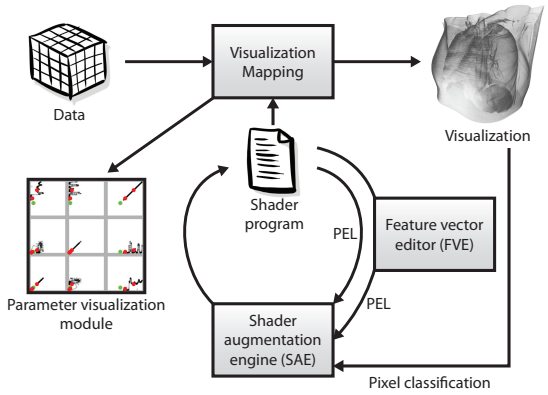


Figure 1: Overview of the visual shader-space exploration.

ify semantics for volume visualization. McDonnel and Elmqvist [12] present the concept of using the GPU for information visualization. They propose an interface for creating visualizations on the GPU without shader-programming knowledge. Jankun-Kelly and Ma [9] propose an interface for parameter space exploration using a spreadsheet-like visualization. The authors show several renderings with different parameter settings which can be used to explore the parameter space. We apply our method to investigate the effects of parameters of particular visualization algorithms on the resulting image. Knowledge gained from this process can be used in the exploration of other datasets using the visualization algorithm.

The goal of our method is to allow comprehensible analysis of visualization algorithms. Our approach is conceptionally similar to some of the mentioned techniques which allows to specify the mapping from data attributes to visual representation in a flexible manner. In contrast, our method allows to modify an existing visualization mapping by generalizing user-defined positive and negative examples in the rendered image, while influence of examined parameters and data attributes is taken into account. This possibility makes our method capable of complex exploration of the shader space, which is difficult to carry out using the existing methods for explicit specification of the visualization mapping.

3 VISUAL SHADER-SPACE EXPLORATION

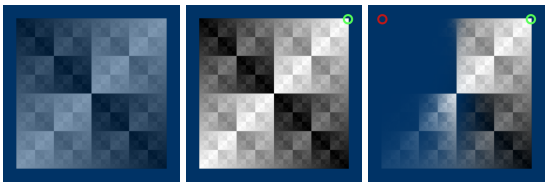
The usual way to explore visualization algorithms is through examination of the results, i.e., rendered images. The effects of various isolated parts of the visualization algorithm are difficult to comprehend by looking at the image space only. However, if the rendered image reveals features of the visualized data, the user can identify areas of the image where the features are visible and areas where they are occluded or not displayed correctly. These areas are positive and negative examples of the visualization-algorithm behaviour. For the purpose of visualization-algorithm exploration, it is interesting to identify variables of the algorithm which could be used for generalizing the behaviour from the positive examples for the whole rendered image. We propose a method for fast determination of whether a particular set of variables can generalize behaviour of the algorithm in a given positive example area for the whole image.

In our visualization-algorithm exploration-method the user interaction is limited to the identification of positive and negative examples in the rendered image, and isolating one or more parameters or data attributes which should be used to generalize those examples. The goal is to help emphasize relationships between various parameters of the visualization algorithms and data features displayed by them. The purpose of the method is to help analyze visualization mappings provided by GPU shader programs and to find ways how the mappings could be enhanced. We call our method visual shader-space exploration (VSSE).

Figure 1 shows the overview of VSSE. An image is created by transforming the data through the visualization mapping. The visualization mapping is implemented by a shader program. Subsequently, the visualization mapping can be modified by the user. In order to modify the visualization mapping, the parameters of the original shader which are to be examined are selected. This can be done either using the Parameter Exploration Language (PEL) which is embedded into the original shader code, or using the Feature Vector Editor (FVE). The FVE also generates PEL code, so the PEL and the FVE can be used simultaneously. The PEL code in the original shader is then parsed with the Shader Augmentation Engine (SAE) and an augmented version of the shader is generated.

The augmented version of the shader provides the original visualization as well as the possibility to select several pixels in the rendered image (i.e., image space of the visualization) where features of interest are visible, and several pixels where they are occluded. We refer to the pixels displaying the features of interest as the positive class, while the other ones constitute the negative class. Subsequently, the way how selected parameters and data attributes are used in the visualization mapping is modified so that the selected examples are generalized to the whole image. This approach constitutes simple yet effective means for analysing the influence of the parameters and the data attributes to features observed in the rendered image.

For every classified pixel a feature vector based on the values of the examined parameters is extracted. The classification of the pixel is then assigned to the extracted feature vector. VSSE provides means to calculate a fuzzy classification, or a membership degree of a



(a) (b) (c)

Figure 2: An example demonstrating the usage of VSSE with a simple shader. In (a), no pixels are classified, opacity is 0.5. In (b), a positive pixel is selected (green circle), opacity is 1 for the whole image. In (c) additionally a negative pixel (red circle) is selected.

feature vector to either the positive or the negative class in any state of the execution of the shader. The membership degree is in the interval $[0, 1]$, where 0 means association with the negative class and 1 means association with the positive class. The value of the membership degree can be freely used in the shader to modify the visualization mapping so that the classification of the rendered pixels is displayed. We also refer to the membership degree as *confidence*.

Figure 2 demonstrates our method on a simple shader displaying a texture. In this case, the feature vectors are composed of texel luminance and coordinates. The confidence is used to modulate the opacity of the texels. In Figure 2(c), two corners of the texture have been classified, one as a positive example, another one as a negative example. The result is a clear separation of the black (top left) from the white (top right) squares. The white square is fully opaque, while the black square is completely transparent. In this example, VSSE shows how texel luminance and its coordinates affects various parts of the rendered texture and how the parts can be separated by setting up one positive and one negative example.

4 PARAMETER EXPLORATION IN SHADER SPACE

In VSSE, the shader-space exploration is carried out by visualizing how parts of the examined shader influence displayed data features. Every pixel of the final image is calculated by a single instance of the visualization shader. The instance is described by a feature vector. The feature vector consists of values extracted during execution of the particular shader instance. Any expression of the shading language can be used for these values. In this way it is possible to extract values of variables or functions at particular positions within the shader program, even in loops or conditional statements. It is also possible to extract multiple feature vectors for a single instance. In that case, every feature vector describes a particular state of the shader program. For instance, in volume rendering, the color of every pixel is determined by compositing several data samples along a ray. For each data sample, a feature vector describing the current sample can be extracted.

The user can classify some of the extracted feature vectors as positive or negative examples. Our method enables the generalization of this classification to all feature vectors extracted during the rendering of the final image. We propose to employ this generalization to modify the visualization mapping. This way, the influence of the selected parameters, data attributes, and intermediate results of the shader on the rendered image can be explored without rewriting the shader. A benefit of our method is also the possibility of taking values of previous executions of the shader into account. Without using our method, this would have to be implemented manually by writing values to the GPU memory and reading them back.

4.1 PEL - Parameter exploration language

We propose a parameter exploration language (PEL), which is the language of shader annotations used for the parameter exploration. The purpose of the PEL is to mark which parameters or data attributes used in the shader program should be examined and how should their effects on the visualized data be displayed. The advantage of using PEL over simply rewriting the shader program in the desired way is that the PEL allows to store multiple values of the examined parameters and data attributes during the shader execution. These values can be stored, classified as positive or negative examples, and used in subsequent executions of the shader program. By using the PEL, this functionality is added to the shader program automatically and there is no need in explicitly creating it.

The PEL can be embedded into the shader source code as shading language comments. Both line and block comments are supported. Comments starting with a dot are intepreted as PEL annotations. They are transformed to regular shader code by the shader augmentation engine. This way, expressions of the PEL and the shading language can be easily combined.

A feature vector can be extracted using the following annotation:

```
//. vector p[0]; ... p[n]; weight
```

This annotation is replaced by code that evaluates the expressions *p[0]*, ... *p[n]* and *weight* and extracts them as the corresponding feature vector.

To allow a flexible specification of how to visualize effects of the examined parameters, the keyword *confidence* can be used in PEL annotations. This keyword

is replaced with a function that evaluates the membership degree for the most recently extracted feature vector. The value represents likeliness of the current shader program state to belong to the positive or the negative class. This value can be arbitrarily displayed by the shader program.

The following listing shows a part of the fragment shader source for the example in Figure 2:

```
vec4 color = texture2D(tex, co.st);
//.vector color.r; co.s; co.t; 1.0
fragColor = vec4(vec3(color.r),
            1.0 /*. - confidence */);
```

In the listing the texel luminance (*color.r*) and coordinates (*co.s, co.t*) define a feature vector. In this example, one feature vector is extracted for every rendered pixel. Some of the pixels are user-classified as negative or as positive. For every other pixel, the confidence of the respective feature vector is calculated and used as opacity (*confidence* keyword) for the pixel.

A feature vector can be extracted in every stage of the shader program. During the execution of the shader, the feature vector may be extracted multiple times for a single pixel. The pixel can be classified as a positive or a negative example by the user. Since the color of the pixel depends on all of the extracted feature vectors, they have to be composited into one feature vector representing the pixel. We refer to it as pixel feature vector. The pixel feature vector is assigned the classification that was specified for the pixel by the user. Various user-selectable composition algorithms, or accumulation types, can be applied. These are defined using *accumulation* keyword of the PEL. The default accumulation type is weighted average using weights assigned to individual extracted feature vectors.

The composition of feature vectors is helpful for applications where the color of the pixel is influenced by multiple data samples, for example volume rendering. The accumulation type should reflect the strategy used for calculating the color of the pixel. Assigning a classification to the pixel feature vector ensures that the positive or negative example is suitably placed in the feature vector space.

The augmented version of the shader can be switched to one of two modes. The first mode is the rendering mode without processing of user input. In this mode, feature vectors are extracted, and the membership function is evaluated for the most recently extracted feature vector. The extracted feature vectors are not composited into pixel feature vectors and no classification is assigned to them. The second mode is the classification mode. In this mode, one pixel and its user-defined classification has to be selected. For the pixel that is being classified, extracted feature vectors are composited into

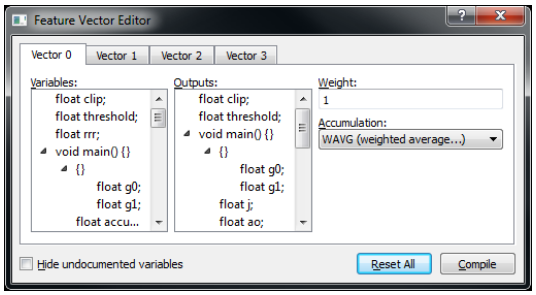


Figure 3: Feature vector editor (FVE).

a classified feature vector at the end of the execution of the shader. The purpose of this design is to enable the host application to use the same shader for rendering as well as user-input processing (classification of feature vectors).

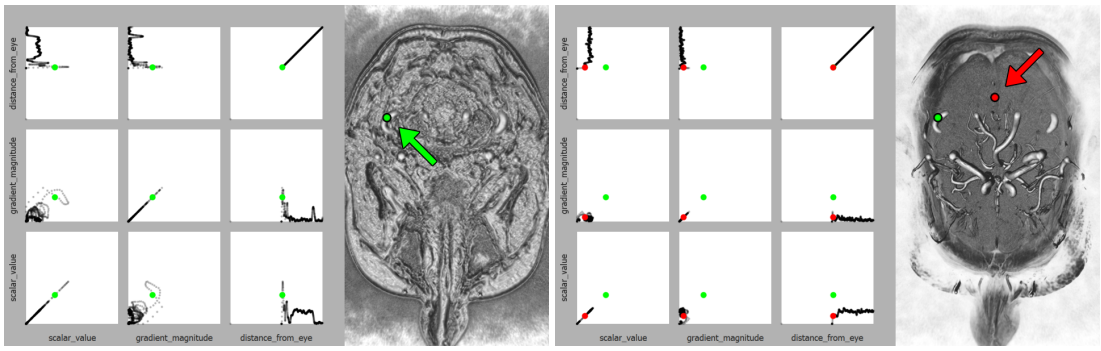
4.2 FVE - Feature vector editor

The PEL offers a high degree of flexibility when designing a parameter exploration scenario for a particular shader program. However, it requires users to actually understand the shader source code in order to create meaningful PEL annotations. To address this issue, we propose a Feature vector editor (FVE). It is a graphical user interface which allows users to create and extract feature vectors from the shader without having to understand it. The FVE interface is shown in Figure 3.

The FVE parses the shader source code and extracts all floating point variables. The extracted variables are then presented to the users so that they can pick any of them to form a feature vector. In the shader code, the formed feature vector is extracted after the last of the selected variables is defined. This means that the variables used for the feature vectors should be initialized with a meaningful value. This is one of the limitations of the FVE. It is not as flexible as using PEL directly, where the user can extract feature vectors from any place in the shader code. Another limitation is that the user can take only variables for the feature vectors, while in PEL any language expression is possible.

The FVE displays the names of the variables as extracted from the shader source code. These names may be confusing for the user and documentation is needed for the effective usage of the FVE. Therefore, we have included two special annotations, *name* and *desc* in the PEL. These annotations are provided for programmers to document individual variables directly in the shader source code so that it is convenient to explore the shader program using FVE. These annotations can change a displayed name of a particular variable and add a description to it. The following listing shows an example:

```
//. name Brightness
//. desc Brightness of the pixel
float br;
```



(a) (b)

Figure 4: A scatterplot matrix displaying feature vectors. Positive vectors are marked with green circles, negative ones with red circles. (a) shows a magnetic resonance angiography dataset of a human brain, where a pixel displaying a vessel was classified as positive (green circle). In (b), a pixel inside the brain matter is classified as negative (red circle), which results in an enhanced view of the vessels. VSSE confirmed that the explored parameters can effectively classify blood vessels in the brain.

The two PEL annotations in the first two lines cause FVE to display *Brightness* instead of *float br;* in the list of variables. Additionally, the FVE displays a pop-up text with the given description on hovering the mouse over the variable.

This mechanism creates a level of abstraction between the shader program source and the parameter exploration. It is the responsibility of the programmer of the shader to provide names and descriptions of variables so that a domain expert can understand them. By using reasonable names and descriptions, the domain expert can explore the shader space without shading-language knowledge, or without a deep understanding of the particular shader program.

We designed our method in such a way that it can benefit domain experts and visualization experts. Visualization experts can use PEL or FVE for analysing their shaders by examining effects of selected parameters. The FVE can be also employed for fast prototyping of new algorithms or visualization mappings derived from the original shader without recompiling the host application.

The visualization expert can prepare the shader for the exploration carried out by the domain expert by adding reasonable domain-specific variable names and descriptions using PEL annotations without actually renaming the variables. The domain expert can then use FVE for exploring the visualization mapping by changing relationships between its parameters and data attributes. The shader program is transparent to the domain expert because the FVE shows the domain-specific names and descriptions of individual variables available for the exploration.

The flexibility of the FVE is not as high as the one provided by PEL directly. In case the FVE is not sufficient in a certain scenario, it is possible to use the PEL annotations together with the FVE. Naturally, in this case knowledge of the shader program is needed.

4.3 Visualization of parameter space

During classification, many potentially multi-dimensional feature vectors are stored in the GPU memory. In addition to visualizing their effects on the rendered image, it is also useful to examine their relationships. For instance, when VSSE is used for identifying suitable parameters for a multi-dimensional transfer function, the parameters of the examined shader program are sequentially chosen for the feature vectors. When a good set of parameters is identified, it is necessary to check whether some of the parameters are correlated. If a pair of the parameters correlates, one of them can be omitted from the designed transfer function.

We introduce a visualization module which is able to depict all positively and negatively classified feature vectors, as well as extracted feature vectors for the currently selected pixels as a context. The visualization module uses a scatterplot matrix, which provides an overview of the bilateral relationships between elements of the vectors.

The visualization module is shown in Figure 4. In this case, feature vectors are composed of following values: scalar data value (*scalar_value*), gradient magnitude (*gradient_magnitude*), distance from the virtual camera (*distance_from_eye*). The goal is to determine whether these three data attributes can be used for effective classification of blood vessels in an MRA scan of a brain. A part of a displayed blood vessel is classified as a positive example for being a vessel. Part of brain matter is classified as a negative example for being a vessel. The opacity of all rendered voxels now depends on how close their attributes are to the specified positive and negative examples.

The user of our method selects examples for the positive and the negative class manually. It is possible to classify several different materials as a positive or negative class. In this case, our method can reveal if the selected variables of the feature vector can provide unified description of these materials.

The visualization module shows the relation of each pair of variables of the feature vector in a scatterplot matrix. For every user-defined classification, multiple values of the variables are extracted and composited into a feature vector. The visualization module can be used to determine the best composition strategy for calculating the feature vectors. In the example of Figure 4, the scatterplots with the *distance to the virtual camera* on one of the axes show peaks on the positively classified pixel, while there are no peaks in the negatively classified one. As the axis of the scatterplots is the distance to the virtual camera, the scatterplots actually show ray profiles. The ray profiles of both the positive and the negative example begin with values close to zero. This means the feature vectors should be composited using a weighted average, otherwise the positive and the negative examples could not be distinguished.

5 FEATURE VECTORS

The user can classify multiple pixels in one session. Therefore, the shader has access to multiple feature vectors associated with either class. At any point of the execution of the shader, this information can be used to calculate the confidence that a most recently extracted feature vector belongs to the positive or the negative class. The confidence is determined by a membership function.

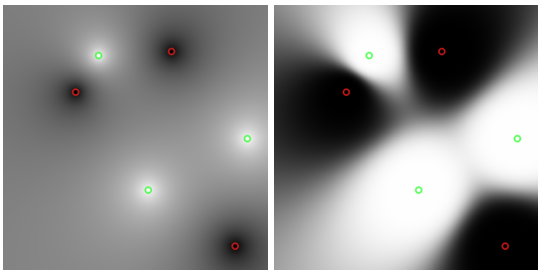
The membership function is used for an automatic classification of pixels that have not been classified by the user as either positive or negative. The function should be smooth and it should not be sensitive to slight changes in its inputs, since these are provided only with a certain accuracy through interaction in image space. For this purpose, Euclidean distances of the classified feature vectors to the pixel feature vector can be employed.

The membership function is defined for pixel feature vectors $\bar{x}_i$ as $f(\bar{x}_i) = c_i$ where c_i is the user defined classification for the feature vector of pixel i. The classification is given as follows:

$$c_i = \begin{cases} 1 & \text{if positive} \\ 0 & \text{if negative} \end{cases} \quad (1)$$

For feature vectors not classified by the user ($\bar{x} \neq \bar{x}_i$), the membership function is defined as follows:

$$f(\bar{x}) = \frac{\sum_{i=1}^{n} (c_i \frac{1}{\|\bar{x}-\bar{x}_i\|^p})}{\sum_{i=1}^{n} (\frac{1}{\|\bar{x}-\bar{x}_i\|^p})} \quad (2)$$



(a) (b)

Figure 5: Visualization of a two-dimensional case of a membership function for (a) $p = 1$ and (b) $p = 5$. Black color means association with the negative class, white color means association with the positive class. Green circles denote positive feature vectors, red circles denote negative ones.

where $\bar{x}_{1..n}$ are the user-classified feature vectors. p is an additional parameter that adjusts softness of the membership function. This fine-tuning is useful when the VSSE is applied to different types of data. Figure 5 gives an example of a two-dimensional feature-vector membership-function with $p = 1$ and $p = 5$.

If there are no user-classified feature vectors yet, the confidence for any feature vector cannot be evaluated. In this case, the membership function is defined as $f(\bar{x}) = 0.5$.

6 IMPLEMENTATION

The system is implemented in C++ using the Qt library. It provides an application programming interface for an easy integration into existing host visualization systems.

The shader augmentation engine is able to enhance shaders written in the GLSL language by replacing PEL annotations with GLSL code. The inserted GLSL code uses the EXT_shader_image_load_store extension for storing and loading feature vectors in OpenGL textures.

When there is a request from the host application to classify a pixel, the host application has to provide the augmented shader with the screen space coordinates and the desired class (positive or negative) of the selected pixel. The augmented shader is then automatically switched to the classification mode. The pixel has to be rendered in this mode, so that its pixel feature vector is calculated, classified, and stored. Other pixels can be rendered in the classification mode as well, but the code for feature vector extraction would be ignored for any pixel with different screen space coordinates from the pixel that was selected (e.g., by mouse clicking on the rendered image).

The parameter visualization module is implemented using GLSL shaders to access extracted feature vectors and render them in a scatterplot matrix. A geometry

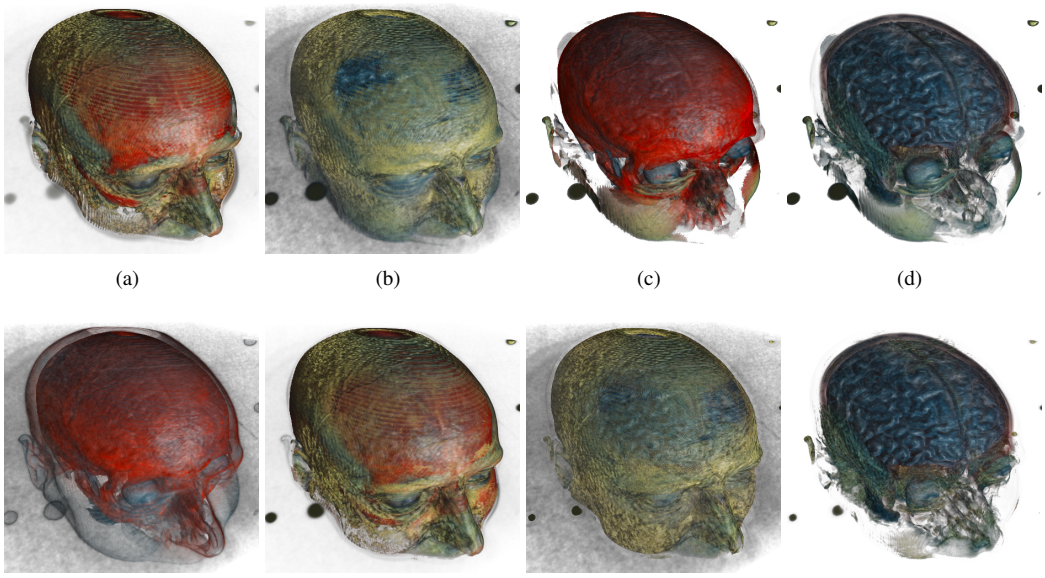


Figure 6: Visualizations of the dual-modality data with positively classified brain and negatively classified occluding tissues for different feature vectors. Feature vectors consist of these attributes: (a) voxCT; (b) gmCT; (c) voxDiff; (d) gmCT, voxDiff; (e) voxMRI; (f) gmDiff, voxMax; (g) gmDiff, gmMax; (h) gmMax, voxDiff

shader is used to create the visualization from the data stored in the GPU memory to minimize data transfer between GPU and CPU.

7 USE CASES

In the following section we present two use cases of VSSE. We apply our method to two distinct shader programs to demonstrate its versatility. In both examples, our method is used to examine an existing algorithm with different goals.

7.1 Volume Rendering

To demonstrate the shader-space exploration-abilities of our method, we employ it for the exploration of a direct volume rendering algorithm. For this use case, we use dual modality data - a co-registered CT and MRI scan of a human head. The MRI data is displayed on the screen, while the CT data is used only to extract additional data attributes. We use VSSE to find out which of the data attributes can be employed to design a transfer function for the effective classification of the brain in this type of data.

We identified several data attributes to be taken under consideration. The data attributes are: scalar value from the MRI data (*voxMRI*), scalar value from the CT data (*voxCT*), gradient magnitude from the MRI data (*gmMRI*), gradient magnitude from the CT data (*gmCT*), difference of the two scalar values (*voxDiff*), difference of the two gradient magnitudes (*gmDiff*), maximum of the two scalar values (*voxMax*), maximum of the two gradient magnitudes (*gmMax*).

Using VSSE, we extract feature vectors for every processed voxel. The feature vectors are formed from various combinations of these data attributes. The method for the composition of the feature vectors is weighted average. The weight for every feature vector is the contribution of the respective voxel to the final pixel color. Therefore, feature vectors composited using this strategy accurately describe the pixels. Finally, we use the confidence value to modulate the opacity of every processed data sample. This means that data samples with feature vectors from the negative class do not contribute to the pixels of the rendered image, thus they do not occlude areas of interest.

Using the FVE we select a set of variables representing the data attributes to form a feature vector. We try various different feature vector setups in order to find most suitable ones. Using a clipping plane, we reveal brain in the visualization of the MRI data. We positively classify one pixel inside the brain. Additionally, three pixels in the skull and other occluding tissues are classified negatively. We use the same user-specified classification for every feature vector setup. Since we classified a pixel displaying the brain as positive, and pixels displaying occluding tissues as negative, the resulting visualization should reveal the brain. If this is not the case, we can conclude that the current set of data attributes would not be suitable for designing a transfer function for the desired visualization mapping. By trying different sets of data attributes, we identify those suitable for classifying brain. Figure 6 shows the results. The best choices are shown in Figure 6(d) and Figure 6(h), where the brain is revealed as expected.

7.2 Image processing

To demonstrate the generality of our method, we show how it can be used to analyse an extension of an image processing algorithm. The goal is to determine whether the extended algorithm is capable of generalising behaviour specified by positive and negative examples.

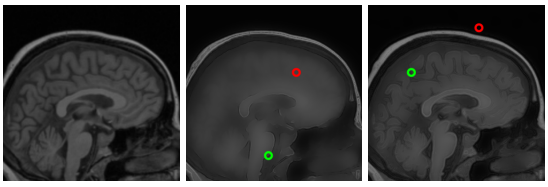
In this example, a shader implementing a bilateral filter [17] is used. The bilateral filter smooths images while preserving edges. It replaces every pixel of the input image with weighted average of surrounding pixels. The weights are determined by a two-dimensional Gaussian function and a color difference between the original and the surrounding pixel. If the difference is higher than a threshold, the weight of the pixel is zero.

The bilateral filter has two parameters: blurring radius and threshold. Higher radius will result in removal of more high-frequency noise, while the threshold determines how strongly should the edges be preserved.

We use our method to explore possibilities of extending the bilateral filter algorithm by using different threshold for every pixel of the rendered image. The goal is to see how the extended algorithm could emphasize features or areas of interest of the images from different application domains. Our method is used to specify these features by classifying several pixels of the image as the positive and the negative examples. Afterwards, it is observed how the classification has been generalized for the rest of the image to evaluate the usefulness of the extended algorithm.

Since the threshold controls how edges are preserved, we apply the Sobel operator to calculate gradient magnitude for every pixel and use it as a feature vector. The gradient magnitude is high for edges and low for flat areas of the image. In this setup, the user can select examples of edges (positive examples) and flat areas (negative examples) by clicking on them. These examples are subsequently generalized for calculating the threshold for each pixel.

Figure 7 shows two different selections of edge examples as well as the original image. In 7(b) only strong edges are visible. In 7(c), a more subtle edge was selected as a positive example. The thresholds were modified so that more edges are visible. Our method shows the potential of the proposed extension of the bilateral filter algorithm for preserving only the edges with a specific significance.



(a) (b) (c)

Figure 7: Using VSSE for exploring possibilities of variable threshold of a bilateral filter. In (a) the original image is shown. In (b) and (c) one positive example of an edge was chosen (green circle) and one negative (red circle). The thresholds for individual pixels are modified so that the examples are generalized for the whole image.

8 DISCUSSION AND LIMITATIONS

Our proposed method is intended for analyzing existing visualization algorithms in order to gain better insight into their inner working. As we show in section 7.1, our method enables rapid exploration of shader space of an visualization algorithm resulting in identification of data-attributes combinations interesting for a specific field. Achieving this goal by simple shader programming would have to be carried out by cumbersome manual evaluation of effectiveness of transfer functions for each visualization mapping. In section 7.2 we show that our method can be employed for a different type of shader analysis as well.

There are several limitations in our method that should be addressed in future work. Currently, the shader augmentation engine is able to parse only GLSL shaders. However, the presented concepts do not depend on using shaders. There is no principal obstacle to provide implementations for other languages implementing the visual mapping.

We have evaluated the performance of the system on a volume rendering shader displaying various datasets. For a dataset of 424x279x190 voxels and a window size of 800x600 pixels, the shader was running at around 120 FPS. After augmenting the shader with the PEL annotations and classifying five pixels using three-dimensional feature vectors, the FPS dropped to approximately 20 FPS. This performance drop is caused by additional GPU memory accesses for extracting values for the feature vectors and reading them back to the CPU for the calculation of the classification of every processed voxel. The timings were obtained with a GeForce GTX 480 graphics card.

9 CONCLUSION

We have proposed visual shader-space exploration, a method for the visual analysis of GPU shader programs. The method enables users to explore effects of various parameters of visualization algorithms. It provides a visual feedback based on user-specified semantic classification of parts of the rendered images. We have implemented the method as a set of C++ classes that are independent of the underlying visualization system.

The implementation could be further enhanced by adding support for different language back-ends (such as OpenCL or CUDA). Various optimizations would

also be possible to improve performance on large datasets, e.g., using caching for the membership functions.

10 ACKNOWLEDGMENTS

The work presented in this paper has been partially supported by the ViMaL project (FWF - Austrian Research Fund, no. P21695) and by the Aktion OE/CZ grant number 64p11.

11 REFERENCES

[1] C. D. Correa and K.-L. Ma. Visibility histograms and visibility-driven transfer functions. *IEEE Transactions on Visualization and Computer Graphics*, 17:192–204, 2011.

[2] P. Crossno and E. Angel. Visual debugging of visualization software: a case study for particle systems. In *Proceedings of the conference on Visualization '99*, VIS '99, pages 417–420. IEEE Computer Society, 1999.

[3] F. de Moura Pinto and C. M. D. S. Freitas. Design of multi-dimensional transfer functions using dimensional reduction. In *Eurographics - IEEE VGTC Symposium on Visualization*, pages 131–138, 2007.

[4] N. Duca, K. Niski, J. Bilodeau, M. Bolitho, Y. Chen, and J. Cohen. A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics*, 24(3):453, 2005.

[5] M. Gavrilescu, M. M. Malik, and M. E. Gröller. Custom interface elements for improved parameter control in volume rendering. In *14th Int. Conf. on System Theory and Control 2010*, pages 219–224, 2010.

[6] M. Gerl, P. Rautek, T. Isenberg, and E. Gröller. Semantics by analogy for illustrative volume visualization. *Computers & Graphics*, 36(3):201–213, 2012.

[7] H. Guo, N. Mao, and X. Yuan. Wysiwyg (what you see is what you get) volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2106 –2114, 2011.

[8] Q. Hou, K. Zhou, and B. Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.*, 28(5), 2009.

[9] T. J. Jankun-Kelly and K.-L. Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[10] T. J. Jankun-Kelly, K. L. Ma, and M. Gertz. A model for the visualization exploration process. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 323–330. IEEE Computer Society, 2002.

[11] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 171–178. IEEE Computer Society, 2004.

[12] B. McDonnel and N. Elmqvist. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1105–1112, 2009.

[13] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Interactive design and debugging of gpu-based volume visualizations. In *Computer Graphics Theory and Applications*, pages 239–245, 2010. short paper.

[14] P. Rautek, S. Bruckner, and E. Gröller. Semantic layers for illustrative volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1336–1343, 2007.

[15] P. Rautek, S. Bruckner, and M. E. Gröller. Interaction-dependent semantics for illustrative volume rendering. *Computer Graphics Forum*, 27(3):847–854, 2008.

[16] M. Strengert, T. Klein, and T. Ertl. A hardware-aware debugger for the opengl shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 81–88. Eurographics Association, 2007.

[17] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. *Sixth International Conference on Computer Vision*, pages 839–846, 1998.

[18] F.-Y. Tzeng, E. Lum, and K.-L. Ma. An intelligent system approach to higher-dimensional classification of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, 2005.

[19] F.-Y. Tzeng and K.-L. Ma. A cluster-space visual interface for arbitrary dimensional classification of volume data. In *Eurographics - IEEE TCVG Symposium on Visualization, 2004*, pages 17–24, 2004.

[20] Y. Wu and H. Qu. Interactive transfer function design based on editing direct volume rendered images. *IEEE Transactions on Visualization and Computer Graphics*, 13(5):1027–1040, 2007.

[21] X. Yuan, N. Zhang, M. X. Nguyen, and B. Chen. Volume cutout. *The Visual Computer (Special Issue of Pacific Graphics 2005)*, 21(8–10):745–754, 2005.